14

The Collections Framework

WHAT YOU WILL LEARN IN THIS CHAPTER

- What sets, sequences, and maps are, and how they work
- The capabilities of the `EnumSet<E>` collection class
- What a `Vector<T>` collection object is and how to use `Vector<T>` objects in your programs
- How to manage `Vector<T>` objects so that storing and retrieving elements is typesafe
- What a `Stack<T>` collection is and how you use it
- How you use the `LinkedList<T>` collections
- How you store and retrieve objects in a hash table represented by a `HashMap<K,V>` object
- How you can generate hashcodes for your own class objects

In this chapter you look at the Java collections framework, which consists of generic types that represent sets of collection classes. These types are defined in the `java.util` package, and they provide you with a variety of ways for structuring and managing collections of objects in your programs. In particular, the collection types enable you to deal with situations where you don't know in advance how many objects you need to store, or where you need a bit more flexibility in the way in which you access an object other than by the indexing mechanism provided by an array.

UNDERSTANDING THE COLLECTIONS FRAMEWORK

The Java collections framework is a set of generic types that you use to create *collection classes* that support various ways for you to store and manage objects of any kind in memory. Recall from Chapter 13 that a collection class is simply a class that organizes a set of objects of a given type in a particular way, such as in a linked list or a pushdown stack. The majority of types that make up the collections framework are defined in the `java.util` package.

Using a generic type for your collections means that you get static checking by the compiler for whatever types of objects you want to manage. This ensures that you do not inadvertently attempt to store objects of the wrong type in a collection. The collections framework includes a professional implementation of a generic linked list type, which is vastly superior to the linked list that you took so much trouble to develop for yourself first as an ordinary class in Chapter 6 and later as a generic type in Chapter 13. However, the effort wasn't entirely wasted because you now have a good idea of how linked lists work and how generic types are defined.

You'll find that the collections framework is a major asset in most of your programs. When you want an array that automatically expands to accommodate however many objects you throw into it, or you need to be able to store and retrieve an object based on what it is rather than using an index or a sequence number, then look no further. You get all this and more in the generic types implemented within the collections framework.

The collections framework involves too much for me to discuss it in complete detail, but you look at how you apply some representative examples of collections that you're likely to need most often. You explore the capabilities provided by the following generic types in detail:

- The `Iterator<T>` interface type declares methods for iterating through elements of a collection, one at a time. You met this interface in the previous chapter.
- The `Vector<T>` type supports an array-like structure for storing objects. The number of objects that you can store in a `Vector<T>` object increases automatically as necessary.
- The `Stack<T>` type supports the storage of objects in a pushdown stack.
- The `LinkedList<T>` type supports the storage of objects in a doubly-linked list, which is a list that you can iterate through forward or backward.
- The `EnumSet<E>` type stores constants of a given enum type, `E`, in a set. You have already used this collection class in file I/O operations.
- The `HashMap<K,V>` type supports the storage of objects of type `V` in a hash table, sometimes called a map. An object is stored using an associated key object of type `K`. To retrieve an object you just supply its associated key.

Let's start by looking in general terms at the various types of collections you can use.

COLLECTIONS OF OBJECTS

In Chapter 13 you put together a generic type that defined a linked list. An object of type `LinkedList<T>` represented an example of a *collection* of objects of type `T`, where `T` could be any class or interface type as long as the type met the conditions required by the collection. A *collection* is the term used to describe any object that represents a set of objects grouped together and organized in a particular way in memory. A class that defines collections of objects is often referred to as a *container class*. A linked list is just one of a number of ways of organizing objects together in a collection.

There are three main groups of collections that organize objects in different ways, called *sets*, *sequences*, and *maps*. Let's first get an understanding of how these kinds of collections work in principle and then look at the specific classes that implement versions of these. One point I'd like to emphasize about the following discussion is that when I talk about a collection of objects, I mean a collection of *references* to objects. In Java, collections store references only—the objects themselves are external to the collection.

Sets

A *set* is probably the simplest kind of collection you can have. Here the objects are not ordered in any particular way at all, and objects are simply added to the set without any control over where they go. It's a bit like putting things in your pocket—you just put things in and they rattle around inside your pocket in no particular order but at least you know roughly where they are. Figure 14-1 illustrates the idea of a set.

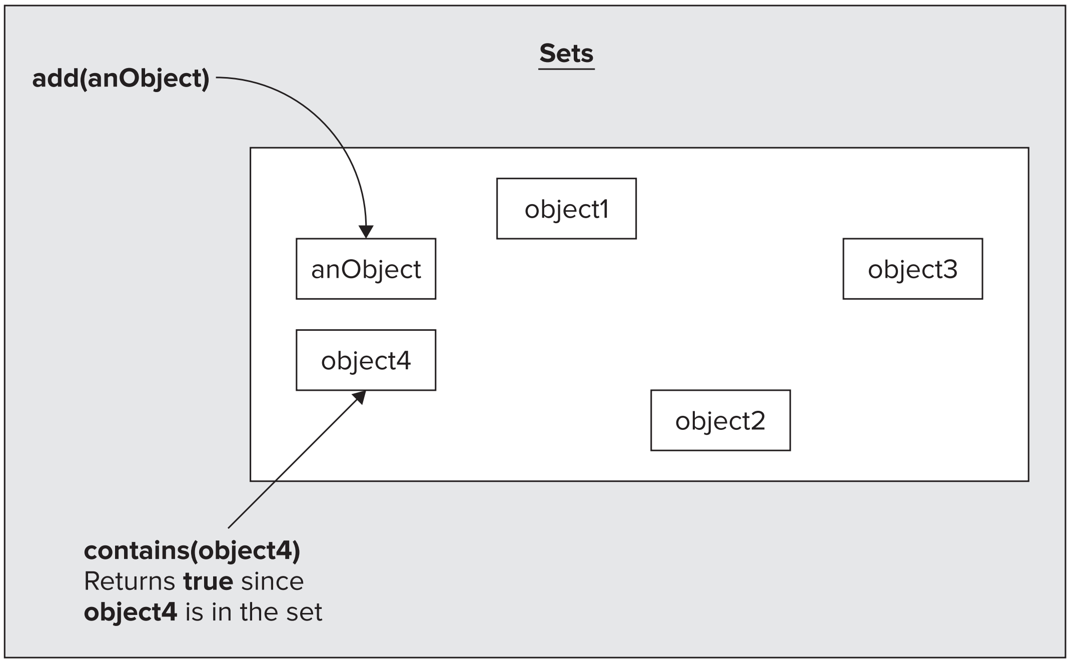


FIGURE 14-1

You can add objects to a set and iterate over all the objects in a set. You can also check whether a given object is a member of the set or not. For this reason you cannot have duplicate objects in a set—each object in a set must be unique. Of course, you can remove a given object from a set, but only if you know what the object is in the first place—in other words, if you have a reference to the object in the set.

There are variations on the basic set that I have described here. For example, sets can be ordered, so objects added to a set are inserted into a sequence ordered according to some comparison criterion. Such sets require that the objects to be stored are of a type that implements the `Comparable<T>` interface.

Sequences

The linked list that you have already explored to some extent is an example of a more general type of collection called a *sequence*. A primary characteristic of a sequence is that the objects are stored in a linear fashion, not necessarily in any particular order, but organized in an arbitrary fixed sequence with a beginning and an end. This contrasts with the set discussed in the previous section, where there may be no order at all. An ordinary array is basically another example of a sequence, but it is more limited than the equivalent collection because it holds a fixed number of elements.

Collections generally have the capability to expand to accommodate as many elements as necessary. The `Vector<T>` type, for example, is an example of a sequence that provides similar functionality to an array, but also has the capability to accommodate as many new elements as you want to add to it. Figure 14-2 illustrates the organization of objects in the various types of sequence collections that you have available.

Because a sequence is linear, you are able to add a new object only at the beginning or at the end, or insert a new object following a given object position in the sequence—after the fifth, say. Generally, you can retrieve an object from a sequence in several ways. You can select the first or the last; you can get the object at a given position—as in indexing an array; or you can search for an object identical to a given object by checking all the objects in the sequence either backward or forward. You can also iterate through the sequence backward or forward accessing each object in turn. You didn't implement all these capabilities in the linked list class in Chapter 6, but you could have.

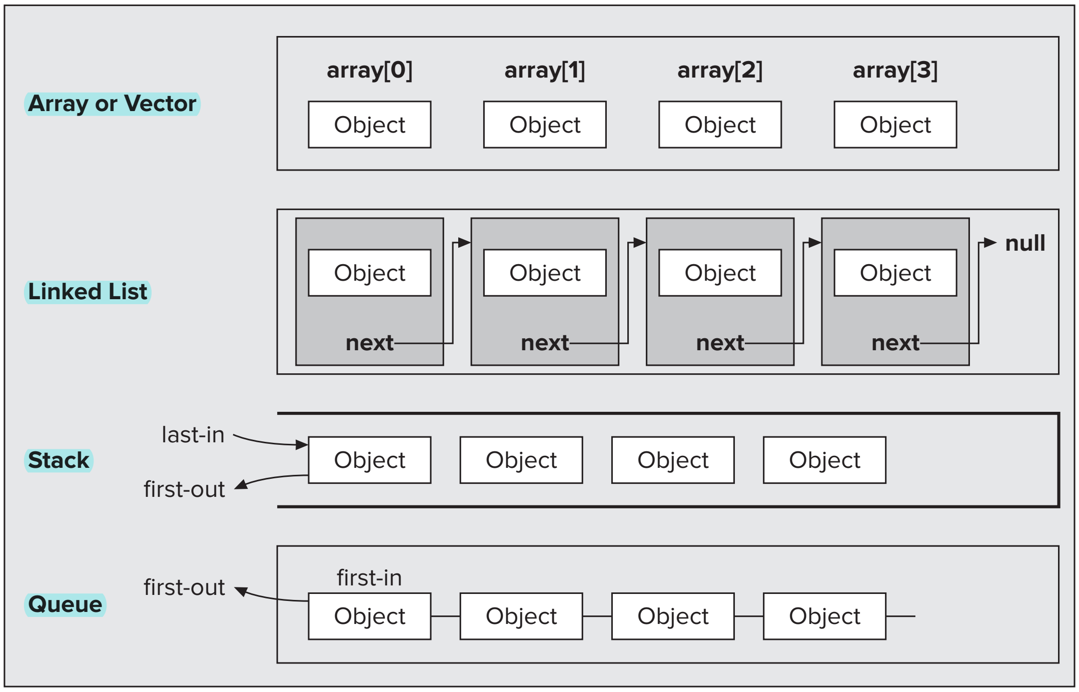


FIGURE 14-2

You have essentially the same options for deleting objects from a sequence as you have for retrieving them; that is, you can remove the first or the last, you can delete the object at a particular position in the sequence, or you can remove an object that is equal to a given object. Sequences have the facility to store several copies of the same object at different places in the sequence. This is not true of all types of collections, as you already know from the outline of a set in the previous section.

A *stack*, which is a last-in, first-out (LIFO) storage mechanism, is also considered to be a sequence, as is a *queue*, which is usually a first-in, first-out (FIFO) mechanism. The Java collections framework implements a range of different queues including a *priority queue* in which elements in the queue are ordered, which implies that FIFO doesn't apply. The elements in a priority queue are in ascending sequence from the head of the queue, so it's more a case of "lowest in, first out."

It's easy to see that a linked list can act as a stack because using the methods to add and remove objects at the end of a list makes the list operate as a stack. Similarly, only adding objects by using the method to add an object to the end of a linked list, and only retrieving objects from the head of the list, makes it operate as a FIFO queue.

Maps

A *map* is rather different from a set or a sequence collection because each entry involves a pair of objects. A map is sometimes referred to as a *dictionary* because of the way it works. Each object that is stored in a map has an associated *key* object, and the object and its key are stored together as a pair. The key determines where the object is stored in the map, and when you want to retrieve an object, you must supply its key—so the key acts as the equivalent of a word that you look up in a regular dictionary. Figure 14-3 shows how a map works.

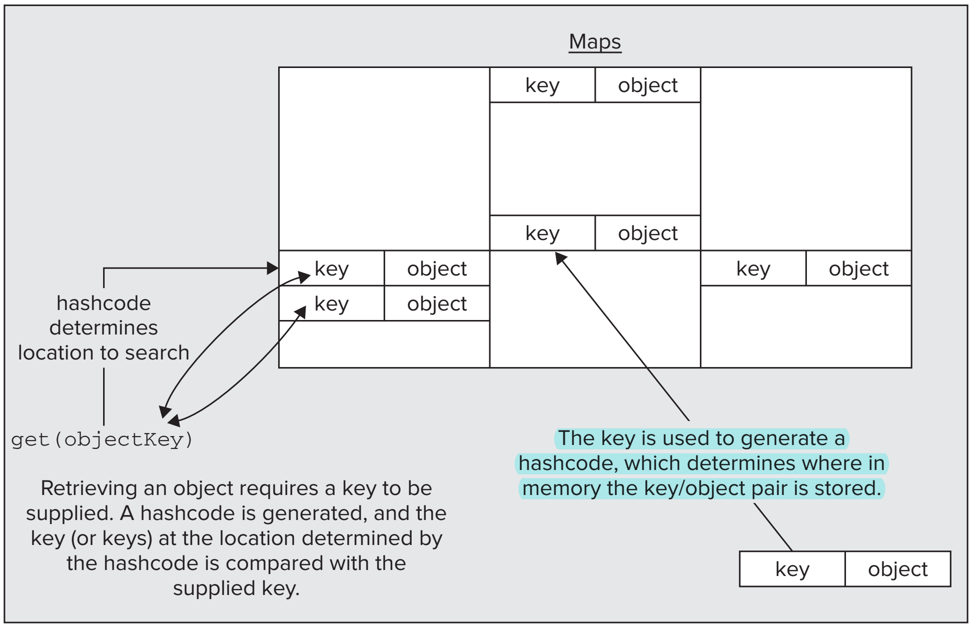

Maps
key
object
key
object
key
object
key
object
key
object
key
object
hashcode determines location to search
get(objectKey)
Retrieving an object requires a key to be supplied. A hashcode is generated, and the key (or keys) at the location determined by the hashcode is compared with the supplied key.
The key is used to generate a hashcode, which determines where in memory the key/object pair is stored.
key
object

FIGURE 14-3

A key can be any kind of object that you want to use to reference the stored objects. Because a key has to uniquely identify an object, all the keys in a map must be different. To put this in context, suppose you were creating a program to provide an address book. You might store all the details of each person—the name, address, phone number, or whatever—in a single object of type `Entry`, perhaps, and store a reference to the object in a map. The key is the mechanism for retrieving objects, so assuming that all names are different, a person's name is a natural choice for the key. Thus the entries in the map in this case would be `Name`/`Entry` pairs. You would supply a `Name` object as the key and get back the `Entry` object corresponding to the key, which might encapsulate data such as the address and/or the phone number. You might also have another map in this application where entries were keyed on the phone number. Then you could retrieve an entry corresponding to a given number. Of course, in practice, names are not unique—hence, the invention of such delightful attachments to the person as Social Security numbers.

Hashing

Where a key/object pair is stored in a map is determined from the key by a process known as *hashing*. Hashing processes the key object to produce an integer value called a *hashcode*. A basic requirement for hashing is that you get the same hashcode from repeatedly hashing the same object.

The `hashCode()` method that is defined in the `Object` class produces a hashcode of type `int` for an object based on the object's location in memory. The hashcode for a key is typically used to calculate an offset from the start of the memory that has been allocated within the map for storing objects, and the offset determines the location where the key/object pair is stored.

Ideally the hashing process should result in values that are uniformly distributed within a given range, and every key should produce a different hashcode. In general, this may not be the case. However, there are ways of dealing with hashcodes that don't turn out to be ideal, so it is not a major problem. The implementations for map collections usually have provision for dealing with the situation where two or more different key objects produce the same hashcode. I explain keys and hashcodes in a little more detail later in this chapter when I discuss using maps.

Now let's look at how you can move through the objects in a collection.

ITERATORS

In the `LinkedList<T>` class that you developed in Chapter 13 you implemented the `Iterable<>` interface for getting the objects from the list. This resulted in your `LinkedList<T>` type being able to make an *iterator* available. As you know, an iterator is an object that you can use once to retrieve all the objects in a collection one by one. Someone dealing cards from a deck one by one is acting as an iterator for the card deck—without the shuffle, of course. Implementing the `Iterable<T>` interface was a much better approach to accessing the members of a list than the technique that you originally implemented, and it made the collection usable with a collection-based `for` loop. An *iterator* is a standard mechanism for accessing elements in a collection.

NOTE *It is worth noting at this point that Java also provides something called an enumerator that is defined by any class that implements the* `java.util.Enumeration<T>` *generic interface type. An enumerator provides essentially the same capability as an iterator, but it is recommended in the Java documentation that you use an iterator in preference to an enumerator for collections. There's nothing particularly wrong with enumerators — it's just that the* `Iterator<T>` *interface declares an optional* `remove()` *method that the* `Enumeration<T>` *interface does not, and the methods in the* `Iterator<T>` *interface have shorter names than those in the* `Enumeration<T>` *interface, so code that uses them is less cluttered.*

Any collection object that represents a set or a sequence can create an object of type `java.util.Iterator<>` that behaves as an iterator. Types representing maps do not have methods for creating iterators. However, a map class provides methods to enable the keys or objects, or indeed the key/object pairs, to be viewed as a set, so you can then obtain an iterator to iterate over the objects in the set view of the map.

An `Iterator<>` object encapsulates references to all the objects in the original collection in some sequence, and they can be accessed one by one using the `Iterator<>` interface methods that you saw in the previous chapter. In other words, an iterator provides an easy way to get at all the objects in a collection one at a time. Just to remind you, the basic mechanism for using an iterator in Java is illustrated in Figure 14-4.

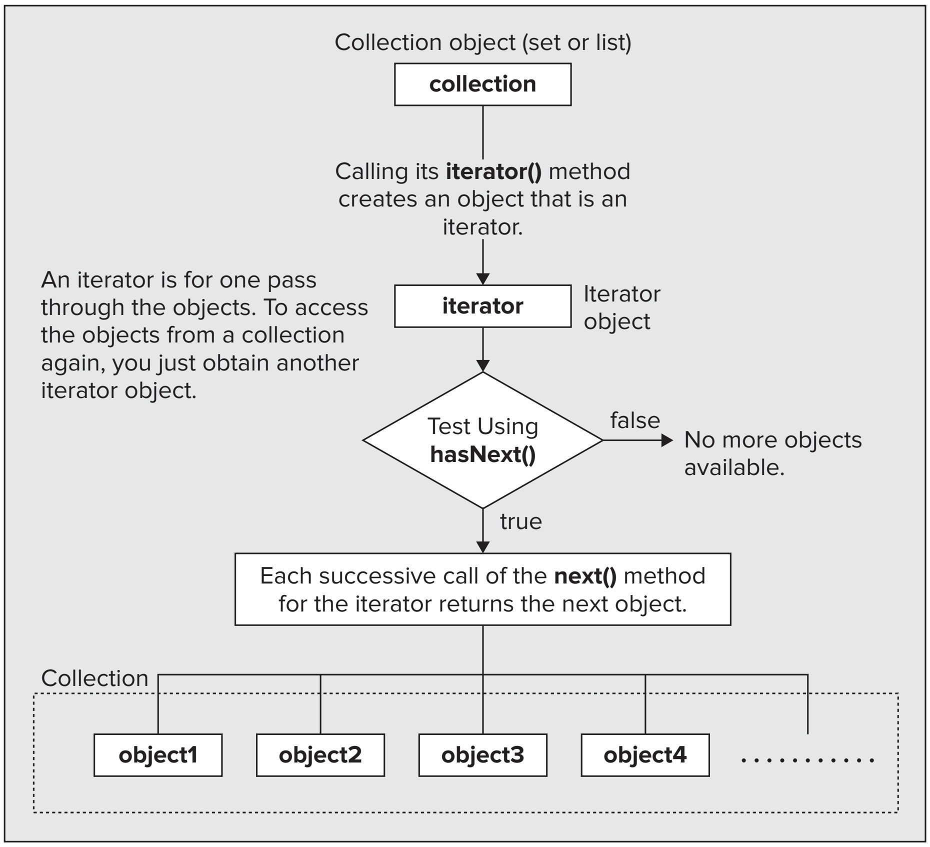

Collection object (set or list)
collection
Calling its iterator() method creates an object that is an iterator.
An iterator is for one pass through the objects. To access the objects from a collection again, you just obtain another iterator object.
iterator
Iterator object
Test Using hasNext()
false
No more objects available.
true
Each successive call of the next() method for the iterator returns the next object.
Collection
object1
object2
object3
object4
....

FIGURE 14-4

As you saw in the previous chapter, the `Iterator<T>` interface declares three methods: `next()`, `hasNext()`, and `remove()`.

> ***NOTE*** *Don't confuse the* `Iterable<T>` *interface with the* `Iterator<T>` *interface.* `Iterable<T>` *defines a method that returns a reference of type* `Iterator<T>;` *in other words, a class that implements* `Iterable<T>` *can make an iterator available.* `Iterator<T>` *defines the methods that enable the iterator mechanism.*

Calling the `next()` method for an iterator returns successive objects from the collection starting with the first, so you can progress through all the objects in a collection very easily with a loop such as the following:

```
MyClass item;                           // Stores an object from the collection
while(iter.hasNext()) {                 // Check that there's another
  item = iter.next();                   // Retrieve next object
  // Do something with item...
}
```

This fragment assumes that `iter` is of type `Iterator<MyClass>` and stores a reference to an object obtained from whatever collection class you were using.

Most collection classes provide an `iterator()` method that returns an iterator for the current contents of the collection. The `next()` method returns an object as the original type so there's no need for casting. As long as the `hasNext()` method returns `true`, there is at least one more object available from the iterator. When all the objects have been accessed, the `hasNext()` method will return `false`. You must obtain a new iterator each time you need to go through the objects in a collection because an iterator is a "use once" object.

The iterator that you have seen here is a single-use one-way street—you can go through the objects in a collection one at a time, in sequence, once, and that's it. This is fine for many purposes but not all, so you have other possibilities for accessing the entire contents of a collection. You can access the objects in any collection that implements the `Iterable<T>` interface directly using the collection-based `for` loop. If this is not enough, there's another kind of iterator that is more flexible than the one you have seen—called a *list iterator*.

List Iterators

The `java.util.ListIterator<T>` interface declares methods that you can use to traverse a collection of objects backward or forward. You don't have to choose a particular direction either. You can change from forward to backward and *vice versa* at any time, so an object can be retrieved more than once.

The `ListIterator<>` interface extends the `Iterator<>` interface type so the iterator methods you have already seen and used still apply. The additional methods that the `ListIterator<>` interface declares for moving through a collection are:

- `int nextIndex()`: Returns the index of the object that is returned by the next call to `next()` as type `int`, or returns the number of elements in the list if the `ListIterator<>` object is at the end of the list.
- `T previous()`: Returns the previous object in sequence in the list. You use this method to run backward through the list.
- `boolean hasPrevious()`: Returns `true` if the next call to `previous()` returns an object.
- `int previousIndex()()`: Returns the index of the object that is returned by the next call to `previous()`, or returns −1 if the `ListIterator<>` object is at the beginning of the list.

You can alternate between calls to `next()` and `previous()` to go backward and forward through the list. Calling `previous()` immediately after calling `next()`, or *vice versa*, returns the same element.

In addition to the `remove()` method, a `ListIterator<>` object provides methods that can add and replace objects:

- `void add(T obj)`: Adds the argument immediately before the object that would be returned by the next call to `next()`, and after the object that would be returned by the next call to `previous()`. The call to `next()` after the `add()` operation returns the object that was added. The next call to previous() is not affected. This method throws an UnsupportedOperationException if objects cannot be added, a ClassCastException if the class of the argument prevents it from being added, and IllegalOperationException if there is some other reason why the add cannot be done.

- void set(T obj): Replaces the last object retrieved by a call to next() or previous(). If neither next() nor previous() have been called, or add() or remove() have been called most recently, an IllegalStateException is thrown. If the set() operation is not supported for this collection, an UnsupportedOperationException is thrown. If the class of the reference passed as an argument prevents the object from being stored, a ClassCastException is thrown. If some other characteristic of the argument prevents it from being stored, an IllegalArgumentException is thrown.

Now that you know more about iterators, you need to find out a bit about the collection classes themselves to make use of them.

COLLECTION CLASSES

The classes in java.util that support collections that are sets, lists, queues, and maps include the following:

Collection Classes That Are Sets

- HashSet<T>: Implements a set using HashMap<T> under the covers. Although a set is by definition unordered, there has to be some way to find an object reasonably efficiently. Using a HashMap<T> object to implement the set enables store and retrieve operations to be done in a constant time. However, the order in which elements of the set are retrieved might vary over time.
- LinkedHashSet<T>: Implements a set using a hash table with all the entries in a doubly-linked list. This class can be used to make a copy of any set such that iteration ordering is preserved—something that does not apply to a HashSet<>.
- TreeSet<T>: Implements a set that orders the objects in ascending sequence. This means that an iterator obtained from a TreeSet<T> object provides the objects in ascending sequence. You can also search the set for the closest match to a given item.
- EnumSet<E extends Enum<E>>: Implements a specialized set that stores enum values from a single enum type, E.

Collection Classes That Are Lists

- Vector<T>: Implements a list as an array that automatically increases in size to accommodate as many elements as you need. Objects are stored and retrieved using an integer index. You can also use an iterator to retrieve objects from a Vector<>. The Vector<> type is synchronized—that is, it is well behaved when concurrently accessed by two or more threads. I will discuss threads and synchronization in Chapter 16.
- ArrayList<T>: Implements an array that can vary in size and can also be accessed as a linked list. This provides a similar function to the Vector<T> generic type but is unsynchronized, so it is not safe for use by multiple threads.
- Stack<T>: This class is derived from Vector<T> and adds methods to implement a stack—a last-in, first-out storage mechanism.
- LinkedList<T>>: Implements a linked list. The linked list defined by this class can also be used as a stack or a queue.

Collection Classes That Are Queues

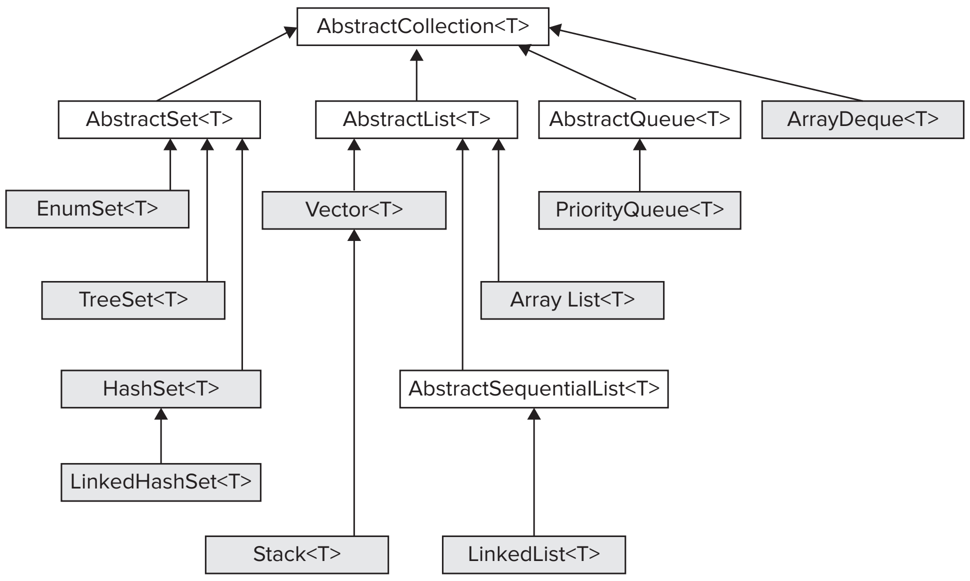
- ArrayDeque<T>: Implements a resizable array as a double-ended queue.
- PriorityQueue<T>>: Implements a priority queue in which objects are ordered in ascending sequence from the head of the queue. The order is determined either by a Comparator<T> object supplied to the constructor for the collection class, or through the compareTo() method declared in the Comparable<T> interface that the object type implements.

Collection Classes That Are Maps

- `Hashtable<K,V>`: Implements a map with keys of type `K` and values of type `V` where all keys must be non-`null`. The key class must implement the `hashcode()` and `equals()` methods to work effectively. This type, like `Vector<T>`, is synchronized, so it's safe for concurrent use by two or more threads.
- `HashMap<K,V>`: Implements a map where objects of type `V` are stored using keys of type `K`. This collection allows `null` objects to be stored and allows a key to be `null` (only one of course, because keys must be unique). The map is not synchronized.
- `LinkedHashMap<K,V>`: Implements a map storing values of type `V` using keys of type `K` with all of its entries in a doubly-linked list. This class can be used to create a copy of a map of any type such that the order of the entries in the copy is the same as the original.
- `WeakHashMap<K,V>`: Implements a map storing values of type `V` using keys of type `K` such that if a reference to a key no longer exists outside the map, the key/object pair is discarded. This contrasts with `HashMap<>` where the presence of the key in the map maintains the life of the key/object pair, even though the program using the map no longer has a reference to the key and therefore cannot retrieve the object.
- `IdentityHashMap<K,V>`: Implements a map storing values of type `V` using keys of type `K` using a hash table where comparisons in searching the map for a key or a value compares references, not objects. This implies that two keys are equal only if they are the same key. The same applies to values. This is a specialized collection class for situations where reference equality is required for both keys and objects.
- `TreeMap<K,V>`: Implements a map storing values of type `V` using keys of type `K` such that the objects are arranged in ascending key order.

This is not an exhaustive list; it is just the classes that are most commonly used. In addition, the `java.util.concurrent` package defines further collection classes that are specifically designed to support concurrent operations by multiple threads.

The generic types representing sets, lists, and queues are related in the manner shown in Figure 14-5.



Relationships between Generic Types Defining Sets, Lists, and Queues

FIGURE 14-5

The shaded boxes identify generic types that you use to define collections. The others are abstract types that you cannot instantiate. All types that define sequences share a common base class, `AbstractCollection<T>`. This class defines methods for the operations that are common to sets, lists, and queues. The operations provided by the `AbstractCollection<T>` class include adding objects to a collection, removing objects, providing an iterator for a collection, and testing for the presence of one or more objects in a collection.

The parameterized types that define maps of various kinds are related as shown in Figure 14-6.

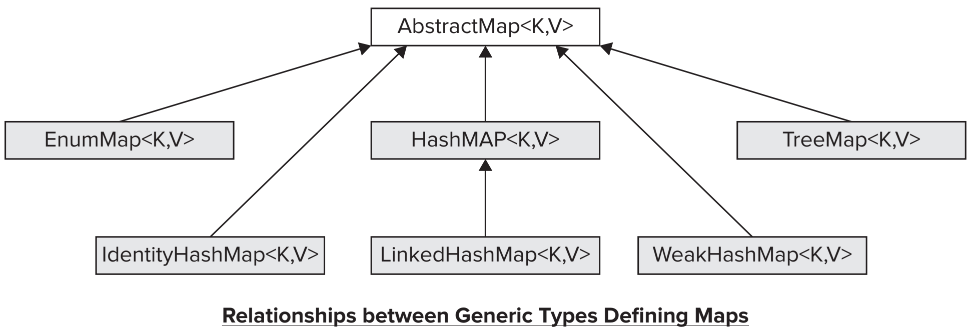


FIGURE 14-6

All the concrete types that define maps have `AbstractMap<K,V>` as a common base class. This provides an outline implementation of operations that apply to maps, thus simplifying the definitions of the more specialized types of maps.

I don't have the space to go into all these collection classes in detail, but to show you some examples of how and where these can be applied, I'll describe the three generic types that you are likely to find most useful: `Vector<T>`, `LinkedList<T>`, and `HashMap<K,V>`. These are representative examples of the most frequently used collections. After you have worked with these, you should have little difficulty with the others. I also explain the `EnumSet<E>` collection class that you have already used in more detail. Before I get into the specifics of using these classes, I introduce the interfaces that they implement, because these define the operations that they support and thus define the ways in which you can apply them.

Collection Interfaces

The `java.util` package defines eight generic collection interface types that determine the methods that you use to work with each type of collection class. These interfaces are related in the manner shown in Figure 14-7.

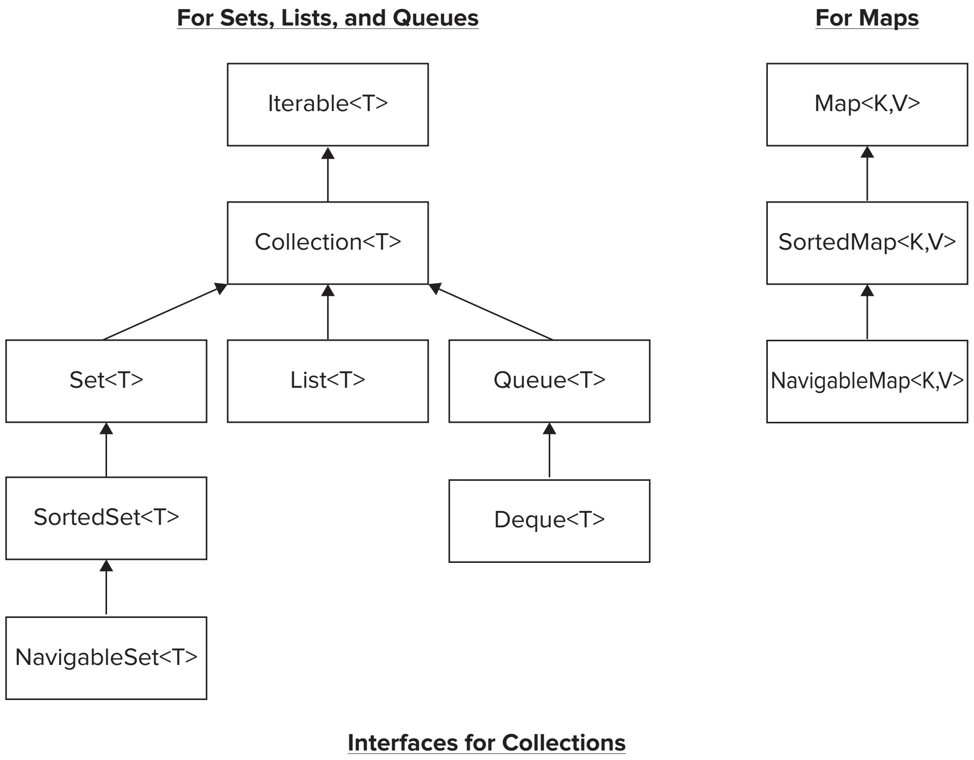


FIGURE 14-7

These are not all the interfaces defined in the `java.util` package, just the ones that relate to the topics in this chapter. You can see that the interfaces for maps have no connection to the interfaces implemented by sets and lists. You can also see that the map interfaces do not implement the `Iterable<T>` interface, so you cannot use the collection-based `for` loop to iterate over the objects in a map. However, the `Map<K,V>` interface declares a `values()` method that returns a collection view of the objects in a map as type `Collection<V>`. You can use the `Collection<V>` reference with a collection-based `for` loop to access the contents of the map because the `Collection<V>` type extends `Iterable<V>`.

There are four basic collection interfaces—the `Set<T>`, `List<T>`, `Queue<T>`, and `Map<K,V>` interfaces—that relate to the fundamental organization of objects in a collection. The first three inherit the members of `Iterable<T>` and `Collection<T>`, so sets, lists, and queues share the characteristics specified by these two interfaces. `SortedSet<T>` and `SortedMap<K,V>` are specialized versions of their respective superinterfaces that add methods for ordering objects within a collection. The `NavigableSet<T>` and `NavigableMap<K,V>` interfaces provide further specialization that defines the capability for finding the closest match for a given search target in a collection. Don't confuse the `Collection<T>` interface with the `Collections` class (with an *s*) that you will see later.

These interfaces are implemented by the classes from the `java.util` package that I have introduced, as shown in Table 14-1:

TABLE 14-1: java.util Package Classes

INTERFACE TYPE	IMPLEMENTED BY
`Set<T>`	`HashSet<T>, LinkedHashSet<T>, EnumSet<T>`
`SortedSet<T>`	`TreeSet<T>`
`List<T>`	`Vector<T>, Stack<T>, ArrayList<T>, LinkedList<T>`
`Queue<T>`	`PriorityQueue<T>, LinkedList<T>`
`Deque<T>`	`LinkedList<T>, ArrayDeque<T>`
`Map<K,V>`	`Hashtable<K,V>, HashMap<K,V>, LinkedHashMap<K,V>, WeakHashMap<K,V>, IdentityHashMap<K,V>, EnumMap<K extends Enum<K>,V>`
`SortedMap<K,V>`	`TreeMap<K,V>`
`NavigableMap<K,V>`	`TreeMap<K,V>`

The `LinkedList<T>` type implements the `List<T>`, the `Queue<T>`, and the `Deque<T>` interfaces, so it really does have multiple personalities in that you can regard a `LinkedList<T>` object as a linked list, as a queue, or as a double-ended queue.

Keep in mind that any collection class object that implements the `Collection<T>` interface can be referenced using a variable of type `Collection<T>`. This means that any of the list or set collections can be referenced in this way; only the map class types are excluded (but not entirely, as you can obtain a set from a map, and the classes implementing a map can provide a view of the values stored as a `Collection<T>` reference). You see that using a parameter of type `Collection<T>` is a standard way of passing a list or a set to a method.

These interfaces involve quite a large number of methods, so rather than go through them in the abstract, let's see them at work in the context of some specific collection class types.

USING ENUMSET

The `EnumSet<E>` collection class enables you to store one or more enumeration constants of a given type, `E`. A primary use for this collection class is as you saw in the context of file I/O. It provides a convenient way of packaging an arbitrary number of enumeration constants so they can be passed to a method as a single argument. There are no constructors for this class. You create an `EnumSet<E>` object by using one of the static factory methods that the class defines.

NOTE *A factory method is a static method that is not a constructor, but nonetheless creates objects. A factory method can provide advantages over a constructor in some situations. For example, it makes it possible to limit the number of objects that may be created. It allows caching of objects so objects may be reused. It also allows a reference to be returned that is an interface type that the object implements. You saw this with static methods in the* `Files` *class.*

The static `of()` method in the `EnumSet<E>` generic class is the factory method that you have already used, and the one you are likely to use most often. It comes in six overloaded versions. Five accept one, two, three, four, or five arguments of type `E` to be stored in the collection. The sixth method uses a varargs parameter list to permit an arbitrary number of arguments of type `E`. The reason for implementing the `of()` method as a set of overloads like this is to maximize efficiency when you put a small number of enumeration constants in a set. The varargs parameter list slows things down, and the overloads for one to five arguments ensure that performance is excellent in the majority of situations.

Assuming you have a static import statement for the constants from the `StandardOpenOption` enumeration, you could define an `EnumSet<>` object like this:

```
EnumSet<StandardOpenOption> set = EnumSet.of(CREATE,WRITE);
```

The more common usage is to create an `EnumSet<>` object in the argument expression, like this:

```
WritableByteChannel fileOut = Files.newByteChannel(
                      file, EnumSet.of(WRITE,  CREATE, TRUNCATE_EXISTING));
```

You have seen this usage several times before in Chapter 10.

After you have the `EnumSet<StandardOpenOption>` object, `set`, you can create another `EnumSet<>` object that represents the set of constants that are not in `set`:

```
EnumSet<StandardOpenOption> setComplement = EnumSet.complementOf(set);
```

`setComplement` contains all the constants from the `StandardOpenOption` enumeration that are not in `set`.

You might want to create an empty set that can hold enumeration constants of a given type:

```
EnumSet<StandardOpenOption> options = EnumSet.noneOf(StandardOpenOption.class);
```

The static `noneOf()` method creates an empty set that can hold constants of the type you specify as the argument. The expression `StandardOpenOption.class` specifies the class type for the `StandardOpenOption` enumeration. You can test for an empty set by calling the `isEmpty()` method for the object.

To add a constant to an existing set, you use the `add()` method:

```
options.add(READ);
```

This adds the `StandardOpenOption.READ` constant to `options`.

The static `copyOf()` method returns a set that is the copy of its argument. The static `allOf()` method returns a set that contains all of the constants of the enumeration type you specify as the argument. For example:

```
EnumSet<StandardOpenOption> optionsCopy = EnumSet.copyOf(options);
EnumSet<StandardOpenOption> allOptions = EnumSet.allOf(StandardOpenOption.class);
```

The `optionsCopy` variable references a new set identical to `options`. The `allOptions` variable references a set containing all the constants from the `StandardOpenOption` enumeration.

Next, I'll present a collection class, which you will note is close to the notion of an array, as you've now become very familiar with arrays.

ARRAY COLLECTION CLASSES

There are two collection classes that emulate arrays: `Vector<T>` and `ArrayList<T>`. Both types define a sequence collection of elements of any type `T`. They work rather like an array, but with the additional feature that they can grow automatically when you need more capacity. They also implement the `List<T>` interface, so you can also access the contents of containers of these types as a list.

These collection classes are similar in function and operation. They have the same base class and implement the same set of interfaces, so understanding the differences between them is useful.

- `Vector<T>` is synchronized so it is safe for concurrent use by more than one thread. `ArrayList<T>` is not synchronized so you should not allow it to be accessed by more than one thread. You can convert an `ArrayList<T>` into a synchronized list using the static `synchronizedList()` method in the `Collections` class.
- You can specify how a `Vector<T>` object should increase its capacity when it is necessary whereas with an `ArrayList<T>` you cannot. You can have some control over how an `ArrayList<T>` increases in capacity by using its `ensureCapacity()` method that sets a given capacity.
- You can discover the current capacity of a `Vector<T>` by calling its `capacity()` method. You have no way to know the current capacity of an `ArrayList<T>` unless you have set its capacity explicitly.
- `ArrayList<T>` is faster in general than a `Vector<T>` because it does not include the overhead for synchronized operation. However, beware of default increments in capacity with an `ArrayList<T>`.

I describe how you use array collection classes in the context of the `Vector<T>` class. However, except where noted, the methods and usage are the same with the `ArrayList<T>` class.

WARNING *Like arrays, array collection class objects hold object references, not actual objects. To keep the text simple I refer to a collection as holding "objects," rather than saying "references to objects." However, you should keep in mind that all the collection classes hold references.*

Creating a Vector

The default constructor creates an empty `Vector<>` object with the capacity to store objects of the type argument that you supply. The default capacity of a `Vector<>` object is 10 objects, and the capacity doubles when you add an object when the container is full. For example:

```
Vector<String> transactions = new Vector<>();
```

This statement creates an empty vector with a capacity for storing references to 10 `String` objects. You can set the initial capacity of the `Vector<>` object explicitly when you create it by using a different constructor. You just specify the capacity you require as an argument of type `int`. For example:

```
Vector<String> transactions = new Vector<>(100);
```

The object you're defining here will have the capacity to store 100 strings initially. It also doubles in capacity each time you exceed the current capacity. The process of doubling the capacity of a vector when more space is required can be quite inefficient. For example, if you add 7,000 `String` object references to the vector you have just defined, it actually has space for 12,800 object references. The capacity-doubling mechanism means that the capacity is always a value of the form 100×2^n, and the smallest n to accommodate 7,000 references is 128. As each object reference requires 4 bytes (on a 32-bit JVM), you are occupying more than 20K bytes unnecessarily.

You have a way of avoiding this with a `Vector<T>` that is not available with the `ArrayList<T>` class. You can specify the amount by which the vector should be incremented as well as the initial capacity when you create it. Both arguments to the constructor are of type `int`. For example

```
Vector<String> transactions = new Vector<>(100,10);
```

This object has an initial capacity of 100, but the capacity is only increased by 10 elements when more space is required.

NOTE *Why not increment the vector object by one each time then? The process of incrementing the capacity takes time because it involves copying the contents of the vector to a new area of memory. The bigger the vector is, the longer the copy takes, and that affects your program's performance if it happens very often.*

You can create a vector or an array list collection that contains objects from another collection. You pass the collection to the constructor as an argument of type `Collection<>`. Because all the set and list collection classes implement the `Collection<>` interface, the constructor argument can be of any set or list class type, including another `Vector<>`. The objects are stored in the new collection class object in the sequence in which they are returned from the iterator for the argument.

Let's see a vector working.

TRY IT OUT Using a Vector

I take a very simple example here, just storing a few strings in a vector:

Available for download on Wrox.com

```
import java.util.Vector;

public class TrySimpleVector {
  public static void main(String[] args) {
    Vector<String> names = new Vector<>();
    String[] firstnames = { "Jack", "Jill", "John",
                            "Joan", "Jeremiah", "Josephine"};

    // Add the names to the vector
    for(String firstname : firstnames) {
      names.add(firstname);
    }

    // List the contents of the vector
    for(String name : names) {
      System.out.println(name);
    }
  }
}
```

TrySimpleVector.java

If you compile and run this, it lists the names that are defined in the program. The code works just as well with `Vector` replaced by `ArrayList`.

How It Works

You first create a vector to store strings using the default constructor:

```
Vector<String> names = new Vector<>();
```

This vector has the default capacity to store ten references to strings. You copy the references to the `Vector<String>` object, `names`, in the first `for` loop. The `add()` method adds the object to the vector at the next available position.

The second `for` loop iterates over the `String` references in the vector:

```
for(String name : names) {
  System.out.println(name);
}
```

All collection classes that are sequences implement the `Iterable<>` interface so you can always use the collection-based `for` loop to access their contents. Of course, you could also use an iterator. The following code produces the same result as the `for` loop:

```
java.util.Iterator<String> iter = names.iterator();
while(iter.hasNext()) {
     System.out.println(iter.next());
}
```

The `iter` object provides a one-time pass through the contents of the `names` collection. The `boolean` value returned by the `hasNext()` method determines whether or not the loop should continue. The `next()` method returns the object reference as the type that you used to create the vector so no casting is necessary.

You are spoiled for choice when accessing elements because you have a third mechanism you can use. The `get()` method returns the reference at the index position specified by the argument to the method. The argument is a zero-based index, just like an array. To iterate over all the index values for elements in a vector you need to know how many elements are stored in it and the `size()` method supplies this as a value of type `int`. You could access the elements like this:

```
for(int i = 0 ; i < names.size() ; ++i) {
  System.out.println(names.get(i));
}
```

The collection-based `for` loop is the simplest and clearest mechanism for iterating over the contents of a vector. However, the `get()` method is useful for accessing an element stored at a particular index position.

Capacity and Size

Although I said earlier that `Vector<>` and `ArrayList<>` collections work like arrays, you can now appreciate that this isn't strictly true. One significant difference is in the information you can get about the storage space provided. An array has a single measure, its length, which is the count of the total number of elements it can reference. Vectors and array lists have two measures relating to the space they provide: the *capacity* and the *size*. However, the capacity of an `ArrayList<>` object is not accessible.

Obtaining and Ensuring Capacity

The *capacity* is the maximum number of objects that a vector or array list can hold at any given instant. Of course, the capacity can vary over time because when you store an object in a `Vector<>` container that is full, its capacity automatically increases. For example, the `Vector<>` object `transactions` that you defined in the last of the constructor examples earlier had an initial capacity of 100. You also specified the capacity increment as 10. After you've stored 101 objects in it, its capacity is 110 objects. A vector typically contains fewer objects than its capacity.

The `capacity()` method returns the current capacity of a vector as a value of type `int`. For example:

```
int namesMax = names.capacity();       // Get current capacity
```

If this statement follows the definition you have for `names` in the previous example, the variable `namesMax` has the value 10. The `capacity()` method is not available in the `ArrayList<>` class.

You can ensure that a vector or an array list has sufficient capacity for your needs by calling its `ensureCapacity()` method. For example:

```
names.ensureCapacity(150);             // Set minimum capacity to 150
```

If the capacity of `names` is less than 150, it is increased to that value. If it's already 150 or greater, it is unchanged by this statement. The argument you specify is of type `int`. There's no return value.

Changing the Size

When you first create a `Vector<>` object , it is empty. The space allocated is occupied after you've stored references in it. The number of elements you have stored in a `Vector<>` is referred to as its *size*. The size of a `Vector<>` clearly can't be greater than the capacity. As you have seen, you can obtain the size as a value of type `int` by calling the `size()` method. You could use the `size()` method in conjunction with the `capacity()` method to calculate the number of free entries in a `Vector<>` object `transactions` with the following statement:

```
int freeCount = names.capacity() - names.size();
```

Storing an object in a vector increases its size, but you can also change the size of a vector directly by calling a method; this does not apply to an array list. Using the method `setSize()`, you can increase and decrease the size. For example:

```
names.setSize(50);                     // Set size to 50
```

The size of the `names` vector is set to the argument value (of type `int`). If the `names` vector has fewer than 50 elements stored, the additional elements up to 50 are filled with `null` references. If it already contains more than 50 objects, all object references in excess of 50 are discarded. The objects themselves may still be available if other references to them exist.

Looking back to the situation I discussed earlier, you saw how the effects of incrementing the capacity by doubling each time the current capacity was exceeded could waste memory. A `Vector<>` or an `ArrayList<>` object provides you with a direct way of dealing with this—the `trimToSize()` method. This changes the capacity to match the current size. For example:

```
names.trimToSize();                    // Set capacity to size
```

If the size of the `names` vector is 30 when this statement executes, then the capacity is set to 30. Of course, you can still add more objects to the `Vector<>` object, and it grows to accommodate them by whatever increment is in effect.

Storing Objects

The simplest way to store an object is to use the `add()` method, as you did in the last example. To store a name in the `names` vector, you could write the following:

```
names.add(aName);
```

This adds a reference to the object `aName` to the `Vector<>` object called `names`. The new entry is added at the end of the vector, and the size is increased by one. All the objects that were already stored in the vector remain at their previous index position.

You can also store an object at a particular index position using another version of `add()`. The first argument is the index position and the second is the object to be stored. The index must be less than or equal to the size of the vector, which implies that either there is already an object reference at this position, or it is the position at the end of the vector that is next in line to receive one. The index is the same as for an array—an offset from the first element—so you reference the first element using an index value of zero. For example, to insert the object `aName` as the third entry in `names`, you would write the following:

```
names.add(2, aName);
```

The index is of type `int` and represents the position of the new object. The new object, `aName`, is inserted in front of the object that was previously at index position 2, so objects stored at index values equal to or greater than 2 are shifted along so their indexes increase by 1. If you specify an index argument that is negative, or greater than the size of the vector, an `ArrayIndexOutOfBoundsException` is thrown.

To replace an element in a vector or array list you use the `set()` method. This accepts two arguments. The first is the index position where the object specified by the second argument is to be stored. To change the third element in the vector `names` to `newName`, you write:

```
names.set(2, newName);
```

The method returns a reference to the object that was previously stored at this position. This gives you a chance to hang on to the displaced object if you want to keep it. If the first argument is negative, or is greater than or equal to the current size of the vector, an `ArrayIndexOutOfBoundsException` is thrown.

You can add all the objects from another collection, either at the end or following a given index position. For example, to append the contents of a `LinkedList<>` object, `myNamesList`—and here I'm referring to the `java.util.LinkedList<>` type, not the homemade version—to a vector, `names`, you would write:

```
names.addAll(myNamesList);
```

The parameter to the method is of type `Collection<? extends T>`, so because the `names` vector is of type `Vector<String>`, the argument must be of type `Collection<String>`. Here, this implies that `myNamesList` is of type `LinkedList<String>`.

To insert the collection of objects at a particular position , you specify the insertion position as the first argument. So to insert the objects from `myNamesList` in the `names` vector starting at index position `i`, you would write:

```
names.addAll(i, myNamesList);
```

The objects originally at and following index position `i` are all shifted along to make room for the new objects. If the index value passed as the first argument is negative, or is greater than the size of `names`, an `ArrayIndexOutOfBoundsException` object is thrown. Adding a collection increases the size of the vector by the number of objects added.

Retrieving Objects

As you saw in the simple example earlier, if you have the index for an element in a vector, you can obtain the element at that position by using the `get()` method. For the `names` vector you could write the following:

```
String name = names.get(4);
```

This statement retrieves the fifth element in the `names` vector. The return type for the `get()` method is determined by the type argument you used to create the `Vector<>` object.

The `get()` method throws an `ArrayIndexOutOfBoundsException` if the argument is an illegal index value. The index must be non-negative and less than the size of the vector.

You can retrieve the first element in a vector using the `firstElement()` method. For example:

```
String name = names.firstElement();
```

You can retrieve the last element in a vector by using the `lastElement()` method in a similar manner. Neither method is implemented in the `ArrayList<>` class. However, both a vector and an array list have the flavor of a list about them, and if you want to process the objects like a list, you can obtain an iterator.

Accessing Elements in a Vector through a List Iterator

You've already seen how you can obtain all the elements in a vector using an `Iterator<>` object, so I won't repeat it. You can also obtain a `ListIterator` reference by calling the `listIterator()` method:

```
ListIterator<String> listIter = names.listIterator();
```

Now you can go backward or forward though the objects in `names` using the `ListIterator` methods that you saw earlier.

It is also possible to obtain a `ListIterator<>` object that encapsulates just a part of a vector, using a `listIterator()` overload that accepts an argument specifying the index position in the vector of the first element in the iterator:

```
ListIterator<String> listIter = names.listIterator(2);
```

This statement results in a list iterator that encapsulates the elements from `names` from index position 2 to the end. If the argument is negative or greater than the size of `names,` an `IndexOutOfBoundsException` is thrown. Take care not to confuse the interface name `ListIterator`, with a capital *L*, with the method of the same name, with a small *l*.

Additionally, you can retrieve an internal subset of the objects as a collection of type `List<>` using the `subList()` method:

```
List<String> list = names.subList(2, 5); // Extract elements 2 to 4 as a sublist
```

The first argument is the index position of the first element from the vector to be included in the list, and the second index is the element at the upper limit—*not* included in the list. Thus this statement extracts elements 2 to 4, inclusive. Both arguments must be positive. The first argument must be less than the size of the vector and the second argument must not be greater than the size; otherwise, an `IndexOutOfBoundsException` is thrown. If the first argument is greater than the second argument, `IllegalArgumentException` is thrown.

You have lots of ways of using the `subList()` method in conjunction with other methods, for example:

```
ListIterator<String> listIter = names.subList(5, 15).listIterator(2);
```

The call to `subList()` returns a `List<String>` object that encapsulates the elements from `names` at index positions 5 to 14, inclusive. You then call the `listIterator()` method for this object, which returns a list iterator of type `ListIterator<String>` for elements in the `List<String>` collection from index position 2 to the end. This corresponds to elements 7 to 14, inclusive, from the original `names` vector. You can use this iterator to roam backward and forward through elements 7 to 14 from the `names` vector to your heart's content.

Extracting All the Elements from a Vector

A `Vector<>` object provides you with tremendous flexibility in use, particularly with the capability to automatically adjust its capacity. Of course, the flexibility you get comes at a price. There is always some overhead involved when you're retrieving elements. For this reason, there might be times when you want to retrieve the elements contained in a `Vector<>` object as a regular array. The `toArray()` method that both array collections implement does this for you. You typically use the method like this:

```
String[] data = names.toArray(new String[names.size()]);
```

The argument to the `toArray()` method must be an array of the same type or a supertype of the type of elements in the vector. If it isn't, an `ArrayStoreException` is thrown. If the argument is `null`, then a `NullPointerException` is thrown. If the array argument is not large enough to accommodate all the elements in the vector, then a new array is created, and a reference to that is returned. The `toArray()` method here returns an array of type `String[]` containing all the elements from `names` in the correct sequence.

It's worth noting that the `java.util.Arrays` class you first saw in Chapter 3 defines a static parameterized method, `asList()`, that converts an array of a given type, `T`, into a `List<T>` collection. The argument is the array of type `T` that you want to convert, and the reference returned is of type `List<T>`. For example:

```
String[] people = { "Brian", "Beryl", "Belinda", "Barry", "Bill", "Barbara" };
List<String> nameList = java.util.Arrays.asList(people);
```

Note that the `List<>` object that is returned does not have storage independent of the array. The `List<>` object is backed by the array you pass as the argument. From the interface hierarchy that you saw earlier you know that a `List<String>` reference is also a `Collection<String>` reference. You can therefore pass it as an argument to a `Vector<String>` constructor. For example:

```
Vector<String> names = new Vector<>( java.util.Arrays.asList(people));
```

Here you are calling the constructor that accepts an argument of type `Collection<>`. You thus have a way to create a `Vector<>` object containing the elements from a predefined array. Of course, the type of elements in the array must be consistent with the type argument for the vector you are creating.

Removing Objects

You can remove the reference at a particular index position by calling the `remove()` method with the index position of the object as the argument. For example:

```
names.remove(3);
```

This removes the fourth element from the `names` vector. The elements that follow the one that was removed are now at index positions that are one less than they were before, so what was previously the fifth element is now at index position 3. Of course, the index value that you specify must be legal for the vector, so it must be greater than or equal to 0 and less than the size of the vector, or an `ArrayIndexOutOfBoundsException` is thrown. The `remove()` method returns a reference to the object removed, so you can retain a reference to the object after you remove it:

```
String name = names.remove(3);
```

Here you save the reference to the object that was removed in `name`.

Sometimes, you want to remove a particular reference, rather than the reference at a given index. If you know what the object is that you want to remove, you can use another version of the `remove()` method to delete it:

```
boolean deleted = names.remove(aName);
```

This searches the `names` vector from the beginning to find the first reference to the object `aName` and removes it. If the object is found and removed from the vector, the method returns `true`; otherwise, it returns `false`. When you are not sure that the element to be removed is present in the vector, you can test the value returned by the `remove()` method.

Another way to remove a single element is to use the `removeElementAt()` method, which requires an argument specifying the index position of the element to be removed. This is similar to the version of `remove()` that accepts an index as an argument, the difference being that here the return type is `void`. This is because the element is always removed if the index you supply is valid, and an `ArrayIndexOutOfBoundsException` is thrown if it isn't.

The `removeAll()` method accepts an argument of type `Collection<>` and removes elements from the collection that you pass to the method if they are present in the vector. The method returns `true` if the `Vector` object is changed by the operation—that is, if at least one element was removed. You could use this in conjunction with the `subList()` method to remove a specific set of elements:

```
names.removeAll(names.subList(5,15));
```

This removes elements 5 to 14, inclusive, from the `Vector<String>` object `names`, plus any duplicates of those objects that are in the vector.

The `removeRange()` method expects two arguments, an index for the first element to be removed, and an index to one past the last element to be removed. So you could apply this to names like this:

```
names.removeRange(5, 15);
```

This removes elements with index values from 5 to 14 inclusive. This is different from the previous statement in that duplicates of these elements will not be removed.

The `retainAll()` method provides a backhanded removal mechanism. You pass a reference of type `Collection<>` as the argument that contains the elements to be retained. Any elements not in the collection you pass to the method are removed. For example, you could keep the elements at index positions

5 to 14, inclusive, plus any duplicates of these elsewhere in the vector, and discard the rest with the following statement:

```
names.retainAll(names.subList(5,15));
```

The method returns `true` if the vector has been changed—in other words, if at least one element has been removed. The method throws a `NullPointerException` if the argument is `null`.

If you want to discard all the elements in a `Vector`, you can use the `removeAllElements()` method to empty the `Vector` in one go:

```
names.removeAllElements();     // Dump the whole lot
```

This removes all the elements from the `names` vector and sets the size to zero. The `clear()` method that is declared in the `List<>` interface is identical in function so you can use that to empty a vector if you prefer.

With all these ways of removing elements from a `Vector<>` object, there's a lot of potential for ending up with an empty container. It's often handy to know whether or not a vector contains elements, particularly if there's been a lot of adding and deleting of elements. You can determine whether a vector contains elements by calling its `isEmpty()` method. This returns `true` if the vector's size is zero and `false` otherwise.

NOTE *Note that if a* `Vector<>` *or* `ArrayList<>` *object contains only* `null` *references, it doesn't mean the* `size()` *is zero or that the* `isEmpty()` *method returns* `true`*. To empty a container you must actually remove all the elements, not just set the elements to* `null`*.*

Searching for Objects

You get the index position of an object stored in a vector if you pass the object as an argument to the `indexOf()` method. For example:

```
int position = names.indexOf(aName);
```

This searches the `names` vector from the beginning for `aName` using the `equals()` method for the argument, so your `aName` class type needs to have a proper implementation of `equals()` for this to work. The variable `position` contains either the index of the first reference to the object in `names`, or −1 if the object isn't found. The `lastIndexOf()` method works in a similar fashion, but the search is starting from the last element back to the first.

Another version of the `indexOf()` method that is not available for an `ArrayList<>` accepts a second argument specifying the index position where the search for the object should begin. The main use for this arises when an object can be referenced more than once in a vector. You can use the method in this situation to recover all occurrences of any particular object, as follows:

```
String aName = "Fred";                                    // Name to be found
int count = 0;                                            // Number of occurrences
int position = -1;                                        // Search starting index
while(++position < names.size()) {                        // Search with a valid index
  if((position = names.indexOf(aName, position)) < 0) {   // Find next
    break;
  }
  ++count;
}
```

This counts the number of occurrences of `a Name` in the `names` vector. The `while` loop continues as long as the `indexOf()` method returns a valid index value and the index isn't incremented beyond the end of the vector `names`. Figure 14-8 shows how this works.

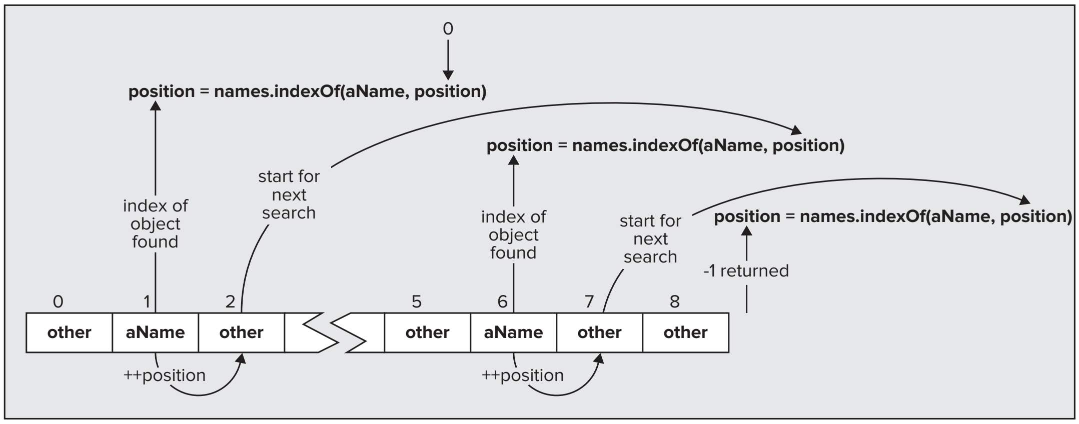


FIGURE 14-8

On each loop iteration, the `indexOf()` method searches `names` from the element given by the index stored in `position`. The initial value of `-1` is incremented in the `while` loop condition, so on the first iteration it is 0. On subsequent iterations, as long as `indexOf()` finds an occurrence of `aName`, the loop condition increments `position` to the next element, ready for the next search. When no further references to the object can be found from the position specified by the second argument, the method returns –1 and the loop ends by executing the `break` statement. If `aName` is found in the last element in the vector at index position `size-1`, the value of `position` is incremented to `size` by the loop condition expression, so the expression is `false` and the loop ends.

When you just want to know whether or not a particular element is stored, and don't really need to know where it is, you can use the `contains()` method that returns `true` if the object you pass as the argument is in the container and returns `false` otherwise. The `Vector<>` and `ArrayList<>` containers also have the `containsAll()` method. You pass a collection to this method as an argument of type `Collection` and the method returns `true` if all the objects in the argument collection are also in the vector or array list.

Applying Vectors

Let's implement a simple example to see a `Vector<>` container working in practice. You write a program to model a collection of people, where you can add the names of the persons that you want in the crowd from the keyboard. You first define a class to represent a person:

```
public class Person {
  // Constructor
  public Person(String firstName, String surname) {
    this.firstName = firstName;
    this.surname = surname;
  }

  @Override
  public String toString() {
    return firstName + " " + surname;
  }

  private String firstName;            // First name of person
  private String surname;              // Second name of person
}
```

The only data members are the `String` members to store the first and second names for a person. You override the inherited version of the `toString()` method to get better output when `Person` class objects are used as arguments to the `println()` method. Now you can define an example with which you can try your skills as a casting director.

TRY IT OUT Creating the Crowd

You can now add a class containing a `main()` method to try storing `Person` objects in a vector. You can call it `TryVector`:

Available for download on Wrox.com

```
import java.util.Vector;
import java.util.ListIterator;
import java.io.*;

public class TryVector {
  public static void main(String[] args) {
    Person aPerson = null;              // A person object
    Vector<Person> filmCast = new Vector<>();

    // Populate the film cast
    while(true) {                       // Indefinite loop
      aPerson = readPerson();           // Read in a film star
      if(aPerson == null) {             // If null obtained...
        break;                          // We are done...
      }
      filmCast.add(aPerson);            // Otherwise, add to the cast
    }

    int count = filmCast.size();
    System.out.println("You added " + count +
                    (count == 1 ? " person":  " people") + " to the cast:");
    // Show who is in the cast using an iterator
    ListIterator<Person> thisLot = filmCast.listIterator();

    while(thisLot.hasNext()) {          // Output all elements
      System.out.println( thisLot.next());
    }
    System.out.println("\nThe vector currently has room for " +
                    (filmCast.capacity() - count) + " more people.");
  }

  // Read a person from the keyboard
  static Person readPerson() {
    // Read in the first name and remove blanks front and back
    String firstName = null;
    String surname = null;
    System.out.println("\nEnter first name or ! to end:");
    try {
      firstName = keyboard.readLine().trim();      // Read and trim a string

      if(firstName.charAt(0) == '!') {             // Check for ! entered
        return null;                               // If so, we are done...
      }

      // Read in the surname, also trimming blanks
      System.out.println("Enter surname:");
      surname = keyboard.readLine().trim();        // Read and trim a string
    } catch(IOException e) {
      System.err.println("Error reading a name.");
      e.printStackTrace();
      System.exit(1);
    }
    return new Person(firstName,surname);
  }

  static BufferedReader keyboard = new BufferedReader(
                                      new InputStreamReader(System.in));
}
```

Directory "TryVector"

With a huge film budget, I got the following output (my input is in bold):

```
Enter first name or ! to end:
Johnny
Enter surname:
Depp

Enter first name or ! to end:
George
Enter surname:
Clooney

Enter first name or ! to end:
Judy
Enter surname:
Dench

Enter first name or ! to end:
Jennifer
Enter surname:
Aniston

Enter first name or ! to end:
!
You added 4 people to the cast:
Johnny Depp
George Clooney
Judy Dench
Jennifer Aniston

The vector currently has room for 6 more people.
```

How It Works

Here you be assembling an all-star cast for a new blockbuster. The `main()` method creates a `Person` variable, which is used as a temporary store for an actor or actress, and a `Vector<Person>` object, `filmCast`, to hold the entire cast.

The `while` loop uses the `readPerson()` method to obtain the necessary information from the keyboard and create a `Person` object. If `!` is entered from the keyboard, `readPerson()` returns `null`, and this ends the input process for cast members.

You output the number of stars entered with these statements:

```
    int count = filmCast.size();
    System.out.println("You added " + count +
                   (count == 1 ? " person":  " people") + " to the cast:");
```

The `size()` method returns the number of objects in the vector, which is precisely what you want. The complication introduced by the conditional operator is just to make the grammar in the output sentence correct.

Just to try it out you output the members of the cast using a `ListIterator<Person>` object. You could do the job just as well with an `Iterator<Person>` object or even a collection-based `for` loop. Using an iterator is still relatively simple:

```
    ListIterator<Person> thisLot = filmCast.listIterator();

    while(thisLot.hasNext()) {       // Output all elements
      System.out.println( thisLot.next());
    }
```

Instead of an iterator, you could have used the `get()` method for the `filmCast` object to retrieve the actors:

```
for(int i = 0 ; i < filmCast.size() ; ++i) {
  System.out.println(filmCast.get(i));
}
```

The collection-based `for` loop is the simplest way of all for listing the contents of the vector:

```
for(Person person : filmCast) {
  System.out.println(person);
}
```

To output the space remaining in the vector, you calculate the difference between the capacity and the size:

```
System.out.println("\nThe vector currently has room for "
           + (filmCast.capacity() - count) + " more people.");
```

This is interesting but irrelevant because the vector accommodates as many stars as you care to enter.

The static `readPerson()` method is a convenient way of managing the input. The input source is the static class member defined by the following statement:

```
static BufferedReader keyboard = new BufferedReader(
                                   new InputStreamReader(System.in));
```

The keyboard object is `System.in` wrapped in an `InputStreamReader` object that is wrapped in a `BufferedReader` object. The `InputStreamReader` object provides conversion of the input from the byte stream `System.in` to characters. The `BufferedReader` object buffers the data read from the `InputStreamReader`. Because the input consists of a series of strings entered one to a line, the `readLine()` method does everything you need. The calls to `readLine()` is in a `try` block because it can throw an `IOException`. The call to the `trim()` method for the `String` object returned by the `readLine()` method removes leading or trailing blanks.

Sorting a Collection

The output from the last example appears in the sequence in which you enter it. If you want to arrange them in alphabetical order you could write your own method to sort `Person` objects in the `filmCast` object, but it is a lot less trouble to take advantage of another feature of the `java.util` package, the `Collections` class—not to be confused with the `Collection<>` interface. The `Collections` class defines a variety of handy static utility methods that you can apply to collections, and one of them happens to be a `sort()` method.

The `sort()` method only sorts lists—that is, collections that implement the `List<>` interface. Obviously there has to be some way for the `sort()` method to determine the order of objects from the list that it is sorting—in your case, `Person` objects. The most suitable way to do this is to implement the `Comparable<>` interface in the `Person` class. As you know, the `Comparable<>` interface declares only one method, `compareTo()`. You saw this method in the previous chapter so you know it returns –1, 0, or +1 depending on whether the current object is less than, equal to, or greater than the argument. If the `Comparable<>` interface is implemented for the type of object stored in a collection, you can just pass the collection object as an argument to the `sort()` method. The collection is sorted in place so there is no return value.

You can implement the `Comparable<>` interface very easily for your `Person` class, as follows:

Available for download on Wrox.com

```
public class Person implements Comparable<Person> {
  // Constructor
  public Person(String firstName, String surname) {
    this.firstName = firstName;
    this.surname = surname;
  }

  @Override
  public String toString() {
    return firstName + " " + surname;
  }

  // Compare Person objects
  public int compareTo(Person person) {
    int result = surname.compareTo(person.surname);
```

```
      return result == 0 ? firstName.compareTo(person.firstName) : result;
    }

    private String firstName;    // First name of person
    private String surname;      // Second name of person
  }
```

Directory "TryVector"

You use the `compareTo()` method in the `String` class to compare the surnames, and if the surnames are equal, the result is determined from the first names.

You can just pass your `Vector<Person>` object to the `sort()` method, and this uses the `compareTo()` method in the `Person` class to compare members of the list.

Let's see if it works for real.

TRY IT OUT Sorting the Stars

You can now add statements to the `main()` method in `TryVector` to sort the cast members:

Available for download on Wrox.com

```
    public static void main(String[] args) {
      // Code as previously...

      // Now sort the vector contents and list it
      Collections.sort(filmCast);
      System.out.println("\nThe cast in ascending sequence is:\n");
      for(Person person : filmCast) {
        System.out.println(person);
      }
    }
```

Directory "TryVector"

You'll need to add the following `import` statement to the `TryVector.java` file:

```
import java.util.Collections;
```

If you run the example with these changes, you get additional output with the cast in alphabetical order. Here's what I got when I entered the same data as last time:

```
Input record and output exactly as before...

The cast in ascending sequence is:
Jennifer Aniston
George Clooney
Judy Dench
Johnny Depp
```

How It Works

Passing the `filmCast` object to the static `sort()` method in the `Collections` class sorts the objects in the vector in place. Like shelling peas!

The `sort()` method is actually a parameterized method so it works for any type that implements the `Comparable<>` interface. The way the type parameter for the method is defined is interesting:

```
static <T extends Comparable<? super T>> void sort(List<T> list)
```

Recall from the discussion of parameterized types in the previous chapter that using a wildcard with the superclass constraint that you see here specifies that the type argument can be any type that implements the `Comparable<>` interface or inherits an implementation from a superclass.

The method parameter is of type `List<T>` rather than `Collection<T>` because the `List<T>` interface provides methods that allow the position where elements are inserted to be determined. It also provides the `listIterator()` method that returns a `ListIterator<T>` object that allows iteration forward and backward through the objects in the collection.

Stack Storage

A stack is a storage mechanism that works on a last-in, first-out basis, which, as you know, is often abbreviated to LIFO. Don't confuse this with FIFO, which is first-in, first-out, or FIFI, which is a name for a poodle. The operation of a stack is analogous to the plate stack you see in some self-service restaurants and is illustrated in Figure 14-9. The stack of plates is supported by a spring that allows the stack of plates to sink into a hole in the countertop so that only the top plate is accessible. The plates come out in the reverse order to the way they went in, so the cold plates are at the bottom, and the hot plates, fresh from the dishwasher, are at the top, which is not so good if you want something chilled.

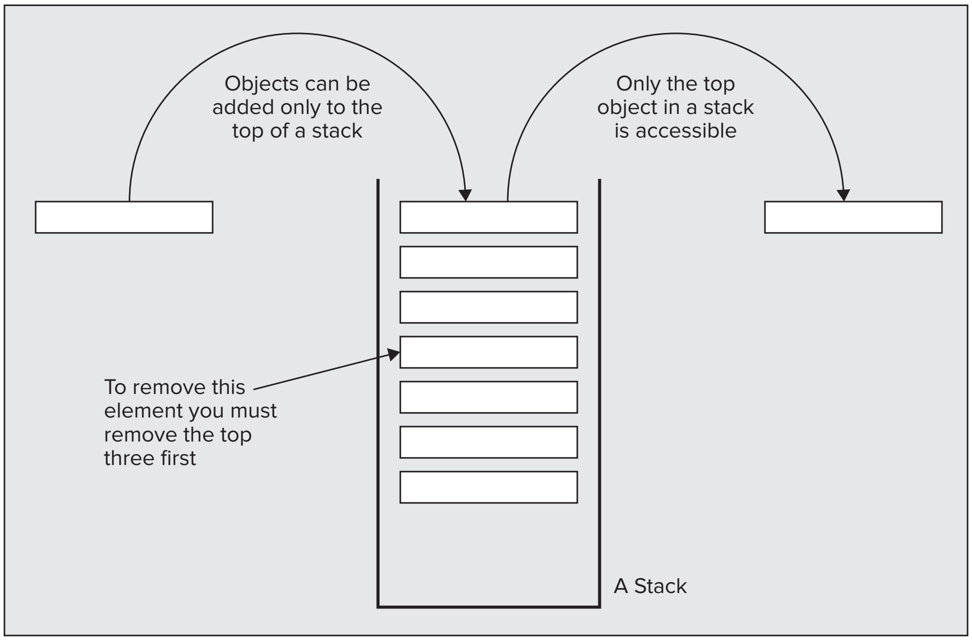

Objects can be added only to the top of a stack
Only the top object in a stack is accessible
To remove this element you must remove the top three first
A Stack

FIGURE 14-9

A stack in Java doesn't have a spring, but it does have all the facilities of a vector because the generic `Stack<>` type is derived from the `Vector<>` type. Of course, you know the `Vector<>` class implements the `List<>` interface so a `Stack<>` object is also a `List<>`.

The `Stack<T>` class adds five methods to those inherited from `Vector<T>`, two of which provide you with the LIFO mechanism; the other three give you extra capabilities. These methods are:

- `T push(T obj)`: Pushes `obj` onto the top of the stack. It also returns the `obj` reference.
- `T pop()`: Pops the object off the top of the stack and returns it. This removes the reference from the stack. If the stack contains no references when you call this method, an `EmptyStackException` is thrown.
- `T peek()`: Returns the object reference at the top of the stack without removing it. Like the previous method, this method can throw an `EmptyStackException`.
- `int search(Object obj)`: Returns a value that is the position of `obj` on the stack. The reference at the top of the stack is at position 1, the next reference is at position 2, and so on. Note that this is quite different from referencing elements in a `Vector<>` or an array, where indexes start at 0. If the object isn't found on the stack, –1 is returned.
- `boolean empty()`: Returns `true` if the stack is empty and returns `false` otherwise.

The only constructor for a `Stack<>` object is the no-arg constructor. This calls the default constructor for the base class, `Vector<>`, so you always get an initial capacity for 10 objects. Because it's basically a vector a stack grows automatically in the same way.

One possible point of confusion is the relationship between the top of a `Stack<>` object and the elements in the underlying `Vector<>` object. Intuitively, you might think that the top of the stack is going to correspond to the first element in the vector, with index 0. If so, you would be totally *wrong*! The `push()` method for a `Stack<>` object is analogous to `add()` for a `Vector<>`, which adds an object to the end of the vector. Thus, the top of the stack corresponds to the end of the vector.

Let's try a `Stack<>` object out in an example so you get a feel for how the methods are used.

TRY IT OUT Dealing Cards

You can use a `Stack<>` object along with another useful method from the `Collections` class to simulate dealing cards from a card deck. You need a way of representing a card's suit and face value or rank. An `enum` type works well because it has a fixed set of constant values. Here's how you can define the suits:

Available for download on Wrox.com

```
public enum Suit {
  CLUBS, DIAMONDS, HEARTS, SPADES
}
```

Directory "TryDeal"

The sequence in which the suits are defined here determines their sort order, so `CLUBS` is the lowest and `SPADES` is the highest. Save this source file as `Suit.java` in a new directory for the files for this example.

You can define the possible card face values just as easily:

Available for download on Wrox.com

```
public enum Rank {
  TWO,   THREE, FOUR, FIVE, SIX,   SEVEN,
  EIGHT, NINE,  TEN,  JACK, QUEEN, KING, ACE
}
```

Directory "TryDeal"

`TWO` is the lowest card face value, and `ACE` is the highest. You can save this as `Rank.java`. You're now ready to develop the class that represents cards. Here's an initial outline:

Available for download on Wrox.com

```
public class Card {
  public Card(Rank rank, Suit suit) {
    this.rank = rank;
    this.suit = suit;
  }

  private Suit suit;
  private Rank rank;
}
```

Directory "TryDeal"

Your `Card` class has two data members that are both enumeration types. One defines the suit and the other defines the face value of the card.

You will undoubtedly need to display a card, so you need a `String` representation of a `Card` object. The `toString()` method can do this for you:

Available for download on Wrox.com

```
public class Card {
  @Override
  public String toString() {
    return rank + " of " + suit;
    }

    // Other members as before...
  }
```

Directory "TryDeal"

The `String` representation of an `enum` constant is the name you assign to the constant, so the `String` representation of a `Card` object with the `suit` value as `CLUBS` and the `rank` value as `FOUR` is `"FOUR of CLUBS"`.

In general, you probably want to compare cards, so you could also make the `Card` class implement the `Comparable<>` interface:

Available for download on Wrox.com

```
public class Card implements Comparable<Card> {

  // Compare two cards
  public int compareTo(Card card) {
    if(suit.equals(card.suit)) {                     // First compare suits
      if(rank.equals(card.rank)) {                   // So check face values
        return 0;                                    // They are equal
      }
      return rank.compareTo(card.rank) < 0 ? -1 : 1;
    } else {                                         // Suits are different
      return suit.compareTo(card.suit) < 0 ? -1 : 1; // Sequence is C<D<H<S
    }
  }

// Other members as before...
}
```

Directory "TryDeal"

You can see the benefit of the `Comparable<>` interface being a generic type. The `Card` class implements the `Comparable<Card>` interface, so the `compareTo()` method works with `Card` objects and no cast is necessary in the operation. The suit first determines the card sequence. If the two cards are of the same suit, then you compare the face values. To compare `enum` values for equality you use the `equals()` method. The `Enum<>` class that is the base for all `enum` types implements the `Comparable<>` interface so you can use the `compareTo()` method to determine the sequencing of `enum` values.

You could represent a hand of cards that is dealt from a deck as an object of type `Hand`. A `Hand` object needs to accommodate an arbitrary number of cards, depending on the game the hand is intended for. You can define the `Hand` class using a `Vector<Card>` object to store the cards:

Available for download on Wrox.com

```
// Class defining a hand of cards
import java.util.Vector;

public class Hand {
  // Add a card to the hand
  public void add(Card card) {
    hand.add(card);
  }

  @Override
  public String toString() {
    StringBuilder str = new StringBuilder();
    boolean first = true;
    for(Card card : hand) {
      if(first) {
        first = false;
      } else {
          str.append(", ");
        }
        str.append(card);
      }
      return str.toString();
    }

    private Vector<Card> hand = new Vector<>();      // Stores a hand of cards
  }
```

Directory "TryDeal"

The default constructor generated by the compiler creates a `Hand` object containing an empty `Vector<Card>` member, `hand`. The `add()` member adds the `Card` object passed as an argument by adding it to the `hand` vector. You also have implemented a `toString()` method in the `Card` class that creates a string that combines the rank name with the suit name. You use the collection-based `for` loop to traverse the cards in the hand and construct a string representation of the complete hand.

It might be good to provide a way to sort the cards in a hand. You could do this by adding a `sort()` method to the `Hand` class:

Available for download on Wrox.com

```
import java.util.Vector;
import java.util.Collections;

public class Hand {
  // Sort the hand
  public Hand sort() {
    Collections.sort(hand);
    return this;
  }

  // Rest of the class as before...
}
```

Directory "TryDeal"

The `Card` class implements the `Comparable<>` interface, so you can use the static `sort()` method in the `Collections` class to sort the cards in the hand. In order to return the current `Hand` object after it has been sorted, the `sort()` method in the class returns `this`. This makes the use of the `sort()` method a little more convenient, as you see when you put the `main()` method together.

You might well want to compare hands in general, but this is completely dependent on the context. The best approach to accommodate this when required is to derive a game-specific class from `Hand`—a `PokerHand` class, for example—and make it implement its own version of the `compareTo()` method in the `Comparable<>` interface.

The last class that you define represents a deck of cards and is able to deal a hand:

Available for download on Wrox.com

```
import java.util.Stack;

public class CardDeck {
  // Create a deck of 52 cards
  public CardDeck() {
    for(Suit suit : Suit.values())
      for(Rank rank : Rank.values())
        deck.push(new Card(rank, suit));
  }

  // Deal a hand
  public Hand dealHand(int numCards) {
      if(deck.size() < numCards) {
        System.err.println("Not enough cards left in the deck!");
        System.exit(1);
      }
      Hand hand = new Hand();
      for(int i = 0 ; i < numCards ; ++i) {
        hand.add(deck.pop());
      }
      return hand;
    }

    private Stack<Card> deck = new Stack<>();
  }
```

Directory "TryDeal"

The card deck is stored as a `Stack<Card>` object, `deck`. In the constructor, the nested `for` loops create the cards in the deck. For each suit in turn, you generate all the `Card` objects for each rank and push them onto the `Stack<>` object, `deck`. The `values()` method for an `enum` type returns a collection containing all the `enum` constants so that's how the loop iterates over all possible suits and ranks.

The `dealHand()` method creates a `Hand` object, and then pops `numCards` `Card` objects off the deck stack and adds each of them to `hand`. The `Hand` object is then returned. At the moment your deck is completely sequenced. You need a method to shuffle the deck before you deal:

Available for download on Wrox.com

```
import java.util.Stack;
import java.util.Collections;

public class CardDeck {
  // Shuffle the deck
  public void shuffle() {
    Collections.shuffle(deck);
  }

  // Rest of the class as before...
}
```

Directory "TryDeal"

With the aid of another `static` parameterized method from the `Collections` class it couldn't be easier. The `shuffle()` method in `Collections` shuffles the contents of any collection that implements the `List<>` interface. The `Stack<>` class implements `List<>` so you can use the `shuffle()` method to produce a shuffled deck of `Card` objects. For those interested in the details of shuffling, this `shuffle()` method randomly permutes the list by running backward through its elements swapping the current element with a randomly chosen element between the first and the current element. The time taken to complete the operation is proportional to the number of elements in the list.

An overloaded version of the `shuffle()` method enables you to supply an object of type `Random` as the second argument, which is used for selecting elements at random while shuffling.

The final piece of the example is a class that defines `main()`:

Available for download on Wrox.com

```
class TryDeal {
  public static void main(String[] args) {
    CardDeck deck = new CardDeck();
    deck.shuffle();

    Hand myHand = deck.dealHand(5).sort();
    Hand yourHand = deck.dealHand(5).sort();
    System.out.println(„\nMy hand is:\n" + myHand);
```

```
        System.out.println(„\nYour hand is:\n" + yourHand);
      }
    }
```

Directory "TryDeal"

I got the following output:

```
My hand is:
THREE of CLUBS, TWO of DIAMONDS, SEVEN of DIAMONDS, FOUR of SPADES, JACK of SPADES

Your hand is:
FOUR of CLUBS, FIVE of CLUBS, TWO of HEARTS, ACE of HEARTS, THREE of SPADES
```

You will almost certainly get something different.

How It Works

Your code for `main()` first creates a `CardDeck` object and calls its `shuffle()` method to randomize the sequence of `Card` objects. You then create two `Hand` objects of five cards with the following statements:

```
Hand myHand = deck.dealHand(5).sort();
Hand yourHand = deck.dealHand(5).sort();
```

The `dealHand()` method returns a `Hand` object that you use to call its `sort()` method. Because the `sort()` method returns a reference to the `Hand` object after sorting, you are able to call it in a single statement like this. The `Hand` object that the `sort()` method returns is stored in the local variable, either `myHand` or `yourHand` as the case may be. The output statements just display the hands that were dealt.

A `Stack<>` object is particularly well suited to dealing cards because you want to remove each card from the deck as it is dealt, and this is done automatically by the `pop()` method that retrieves an object. When you need to go through all the objects in a stack without removing them, you can use a collection-based `for` loop, just as you did for the `Vector<Card>` object in the `toString()` method in the `Hand` class. Of course, because the `Stack<>` class is derived from `Vector<>`, all the `Vector<>` class methods are available for a stack when you need them.

I think you'll agree that using a stack is very simple. A stack is a powerful tool in many different contexts. A stack is often applied in applications that involve syntactical analysis, such as compilers and interpreters—including those for Java.

LINKED LISTS

The `LinkedList<T>` type implements the List<> interface and defines a generalized linked list. You have already seen quite a few of the methods that the class implements, as the members of the `List<>` interface are implemented in the `Vector<>` class. Nonetheless, here I quickly review the methods that the `LinkedList<>` class implements. There are two constructors: a default constructor that creates an empty list and a constructor that accepts a `Collection<>` argument that creates a `LinkedList<>` object that contains the objects from the collection that is passed to it.

To add objects to a list you have the `add()` and `addAll()` methods, exactly as I discussed for a `Vector<>` object. You can also add an object at the beginning of a list using the `addFirst()` method, and you can add one at the end using `addLast()`. Both methods accept an argument of a type corresponding to the type argument you supplied when you created the `LinkedList<>` object, and neither return a value. Of course, the `addLast()` method provides the same function as the `add()` method.

To retrieve an object at a particular index position in the list, you can use the `get()` method, as in the `Vector<>` class. You can also obtain references to the first and last objects by using the `getFirst()` and `getLast()` methods, respectively. To remove an object you use the `remove()` method with an argument that is either an index value or a reference to the object that is to be removed. The `removeFirst()` and `removeLast()` methods do what you would expect.

Replacing an existing element in the list at a given index position is achieved by using the `set()` method. The first argument is the index and the second argument is the new object at that position. The old object is returned, and the method throws an `IndexOutOfBoundsException` if the index value is not within the limits of the list. The `size()` method returns the number of elements in the list.

As with a `Vector<>` object, you can obtain an `Iterator<>` object by calling `iterator()`, and you can obtain a `ListIterator<>` object by calling `listIterator()`. Recall that an `Iterator<>` object enables you only to go forward through the elements, whereas a `ListIterator<>` object enables you to iterate backward and forward.

You could change the `TryPolyLine` example from Chapter 6 to use a `LinkedList<>` collection object rather than your homemade version.

TRY IT OUT Using a Genuine Linked List

Put this example in a new directory, `TryPolyLine`. You can use the `TryPolyLine` class that contains `main()` and the `Point` class exactly as they are, so if you still have them, copy the source files to the new directory. You just need to change the `PolyLine` class definition:

Available for download on Wrox.com

```
import java.util.LinkedList;

public class PolyLine {
  // Construct a polyline from an array of points
  public PolyLine(Point[] points) {
    // Add the  points
    for(Point point : points) {
      polyline.add(point);
    }
  }
  // Construct a polyline from an array of coordinates
  public PolyLine(double[][] coords) {
    for(double[] xy : coords) {
      addPoint(xy[0], xy[1]);
    }
  }

  // Add a Point object to the list
  public void addPoint(Point point) {
    polyline.add(point);               // Add the new point
  }

  // Add a point to the list
  public void addPoint(double x, double y) {
    polyline.add(new Point(x, y));
  }

  // String representation of a polyline
  @Override
  public String toString() {
    StringBuffer str = new StringBuffer("Polyline:");

    for(Point point : polyline) {
      str.append(" " + point);         // Append the current point
    }
    return str.toString();
  }

  private LinkedList<Point> polyline = new LinkedList<>();
}
```

Directory "TryPolyLine"

The class is a lot simpler because the `LinkedList<>` class provides all the mechanics for operating a linked list. Because the interface to the `PolyLine` class is the same as the previous version, the original version of `main()`runs unchanged and produces exactly the same output.

How It Works

The only interesting bit is the change to the `PolyLine` class. `Point` objects are now stored in the linked list implemented by the `LinkedList<>` object, `polyline`. You use the `add()` method to add points in the constructors, and the `addPoint()` methods. Using a collection class makes the `PolyLine` class very straightforward.

I changed the implementation of the second constructor in the `PolyLine` class to illustrate how you can use the collection-based `for` loop with a two-dimensional array:

```
public PolyLine(double[][] coords) {
  for(double[] xy : coords) {
    addPoint(xy[0], xy[1]);
  }
}
```

The `coords` parameter to the constructor is a two-dimensional array of elements of type `double`. This is effectively a one-dimensional array of references to one-dimensional arrays that have two elements each, corresponding to the *x* and *y* coordinate values for a point. Thus, you can use the collection-based `for` loop to iterate over the array of arrays. The loop variable is `xy`, which is of type `double[]` and has two elements. Within the loop, you pass the elements of the array `xy` as arguments to the `addPoint()` method. This method then creates a `Point` object and adds it to the `LinkedList<Point>` collection, `polyline`.

USING MAPS

As you saw at the beginning of this chapter, a *map* is a way of storing data that minimizes the need for searching when you want to retrieve an object. Each object is associated with a key that is used to determine where to store the reference to the object, and both the key and the object are stored in the map. Given a key, you can always go more or less directly to the object that has been stored in the map based on the key. It's important to understand a bit more about how the storage mechanism works for a map, and in particular what the implications of using the default hashing process are. You explore the use of maps primarily in the context of the `HashMap<K,V>` generic class type.

The Hashing Process

The implementation of a map in the Java collections framework that is provided by the `HashMap<K,V>` class sets aside an array in which it stores key and object pairs of type `K` and `V` respectively. The index to this array is produced from the key object by using the hashcode for the object to compute an offset into the array for storing the key/object pair. By default, this uses the `hashCode()` method for the key object. This is inherited in all classes from `Object` so this is the method that produces the basic hashcode unless the `hashCode()` method is redefined in the class for the key. The `HashMap<>` class does not assume that the basic hashcode is adequate. To try to ensure that the hashcode that is used has the characteristics required for an efficient map, the basic hashcode is further transformed within the `HashMap<>` object.

An entry in the table that is used to store key/value pairs is called a *bucket*. The hashcode produced from a key selects a particular bucket in which the key/value pair should be stored. This is illustrated in Figure 14-10.

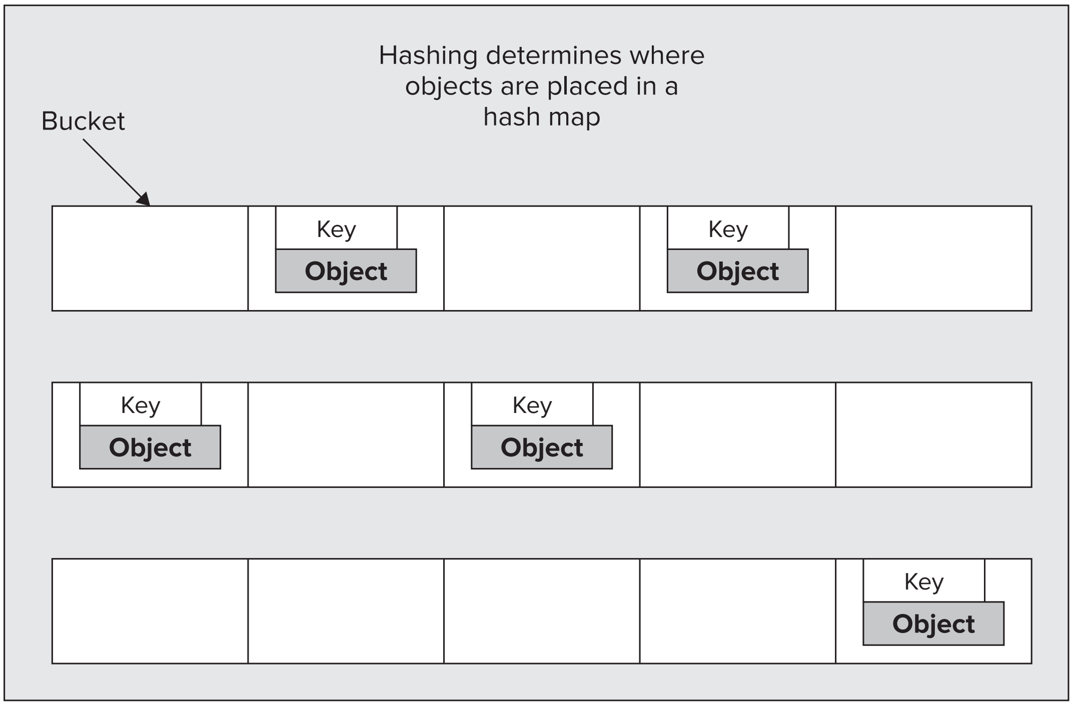


FIGURE 14-10

Note that, although every key must be unique, each key doesn't have to result in a unique hashcode. When two or more keys produce the same hash value, it's called a *collision*. A `HashMap<>` object deals with collisions by storing all the key/object pairs that have the same hash value in a linked list. If this occurs often, it is obviously going to slow down the process of storing and retrieving data. Retrieving an object that resulted in a collision when it was stored is a two-stage process. The key is hashed to find the location where the key/object pair should be. The linked list then has to be searched to sort out the key you are searching on from all the others that have the same hash value. There is therefore a strong incentive to minimize collisions, and the price of reducing the possibility of collisions in a hash table is having plenty of empty space in the table.

The `Object` class defines the `hashCode()` method so any object can be used as a key and it hashes by default. The method as it is implemented in `Object` in Java, however, isn't a panacea. Because it usually uses the memory address where an object is stored to produce the hash value, distinct objects always produce different hash values. In one sense this is a plus, because the more likely it is that a unique hash value is produced for each key, the more efficient the operation of the hash map is going to be. The downside is that different objects that have identical data produce different hash values, so you can't compare them.

This becomes a nuisance if you use the default `hashCode()` method for objects that you're using as keys. In this case, an object stored in a hash map can never be retrieved using a different key object instance, even though that key object may be identical in all other respects. Yet this is precisely what you want to do in many cases.

Consider an application such as a simple address book. You might store map entries keyed on the names of the people to whom the entries relate, and you would want to search the map based on a name that was entered from the keyboard. However, the object representing the newly entered name is inevitably going to be distinct from that used as a key for the entry. Using the former, you won't be able to find the entry corresponding to the name.

The solution to this problem is to create a hashcode from the instance variables of the object. Then, by comparing the hashcode produced from the data members of the new name object with the hashcodes for the name objects used as keys in the hash map, you are able to make a match.

Using Your Own Class Objects as Keys

For objects of one of your own classes to be usable as keys in a hash table, you must override the `equals()` method of the `Object` class. In its default form, `equals()` accepts an object of the same class as an argument and returns a `boolean` value. The `equals()` method is used by methods in the `HashMap<>` class to determine when two keys are equal, so in order to enable the changes discussed in the previous section, your version of this method should return `true` when two different objects contain identical data values.

You can also override the default `hashCode()` method, which returns the hash value for the object as type `int`. The `hashCode()` method is used to generate the value that determines where a key/object pair is located. Your `hashCode()` method should produce hashcodes that are reasonably uniform over the possible range of keys and is generally unique for each key.

Generating Hashcodes

The various techniques for generating hashcodes form a big topic, and I can barely scratch the surface here. How you write the `hashCode()` method for your class is up to you, but it needs to meet certain requirements if it is to be effective. A hashcode is returned by the `hashCode()` method as a value of type `int`. You should aim to return a hashcode that has a strong probability of being unique to the object, and the hashcodes that you produce for the range of different objects that you are working with should be as widely distributed across the range of `int` values as possible.

To achieve the uniqueness, you typically want to combine the values of all the data members in an object to produce the hashcode, so the first step is to produce an integer corresponding to each data member. You must then combine these integers to generate the return value that is the hashcode for the object. One technique you can use to do this is to multiply each of the integers corresponding to the data members by a different prime number and then sum the results. This should produce a reasonable distribution of values that have a good probability of being different for different objects. It doesn't matter which prime numbers you use as multipliers, as long as:

- They aren't so large as to cause the result to fall outside the range of type `int`.
- You use a different one for each data member.

So how do you get from a data member of a class to an integer? Generating an integer for data members of type `String` is easy: you just call the `hashCode()` method for the member. This has been implemented in the `String` class to produce good hashcode values that are the same for identical strings (take a look at the source code if you want to see how). You can use integer data members as they are, but floating-point data members need a bit of judgment. If they have a small range in integer terms, you need to multiply them by a value that's going to result in a unique integer when they are cast to type `int`. If they have a very large range in integer terms you might need to scale them down.

Suppose you intended to use a `Person` object as a key in a hash table, and the class data members were `firstName` and `surname` of type `String` and `age` of type `int`. You could implement the `hashCode()` method for the class as the following:

```
public int hashCode() {
  return 13*firstName.hashCode() + 17*surname.hashCode() + 19*age;
}
```

Wherever a data member is an object of another class rather than a variable of one of the basic types, you need to implement the `hashCode()` method for that class. You can then use that in the computation of the hashcode for the key class.

Creating a HashMap Container

As you saw earlier in this chapter, all map classes implement the `Map<>` interface, so an object of any map class can be referenced using a variable of type `Map<>`. You look in detail at the `HashMap<>` class because it is good for most purposes. There are four constructors to create a `HashMap<K,V>` object:

- `HashMap()` creates a map with the capacity to store a default number of objects. The default capacity is 16 objects, and the default load factor (more on the load factor below) is 0.75.
- `HashMap(int capacity)` creates a map with the capacity to store the number of objects you specify in the argument and a default load factor of 0.75.
- `HashMap(int capacity,float loadFactor)` creates a map with the capacity and load factor that you specify.
- `HashMap(Map<? extends K, ? extends V> map)` creates a map with the mappings, capacity and load factor of the `Map` object passed as the argument.

To create a map using the default constructor, you can write something like this:

```
HashMap<String,Person> theMap = new HashMap<>();
```

This statement creates a `HashMap<>` object that can store `Person` objects with associated keys of type `String`.

The *capacity* for a map is simply the number of key/object pairs it can store. The capacity increases automatically as necessary, but this is a relatively time-consuming operation. The capacity value of the map is combined with the hashcode for the key that you specify to compute the index that determines where an object and its key are to be stored. To make this computation produce a good distribution of index values, you should ideally use prime numbers for the capacity of a hash table when you specify it yourself. For example:

```
HashMap myMap<String,Person> = new HashMap<>(151);
```

This map has a capacity for 151 objects and their keys, although the number of objects stored can never actually reach the capacity. You must always have spare capacity in a map for efficient operation. With too little spare capacity, you have an increased likelihood that keys generate the same bucket index, so collisions become more likely.

The *load factor* is used to decide when to increase the size of the hash table. When the size of the table reaches a value that is the product of the load factor and the capacity, the capacity is increased automatically to twice the old capacity plus one—the plus one ensuring it is at least odd, if not prime. The default load factor of 0.75 is a good compromise, but if you want to reduce it you could do so by using the third constructor:

```
// Create a map with a 60% load factor
HashMap<String,Person> aMap = new HashMap<>(151, 0.6f);
```

This map works a bit more efficiently than the current default, but at the expense of having more unoccupied space. When 90 objects have been stored, the capacity is increased to 303, $(2 \times 151 + 1)$.

Storing, Retrieving, and Removing Objects

Storing, retrieving, and removing objects in a `HashMap<>` is very simple. The methods involved in these operations are the following:

- `V put(K key, V value)`: Stores the object `value` in the map using the key specified by the first argument. `value` replaces any existing object associated with `key`, and a reference to the previous object for the key is returned. If no object was previously stored for `key` or the key was used to store `null` as an object, `null` is returned.
- `void putAll(Map<? extends K,? extends V> map)`: Transfers all the key/object pairs from `map` to the current map, replacing any objects that exist with the same keys.
- `V get(Object key)`: Returns the object stored with the same key as the argument. If no object was stored with this key or `null` was stored as the object, `null` is returned. Note that the object remains in the map.
- `V remove(Object key)`: Removes the entry associated with `key` if it exists and returns a reference to the object. A `null` is returned if the entry does not exist, or if `null` was stored using `key`.

If you attempt to retrieve an object using `get()` and a `null` is returned, it is still possible that a `null` was stored as the object associated with the key that you supplied to the `get()` method. You can determine if this is the case by passing your key object to the `containsKey()` method for the map. This returns `true` if the key is stored in the map.

You should ensure that the value returned from the `put()` method is `null`. If you don't, you may unwittingly replace an object that was stored in the table earlier using the same key. The following code fragment illustrates how you might do that:

```
HashMap<String,Integer> aMap = new HashMap<>();
String myKey = "Goofy";
int value = 12345;
Integer oldValue = null;
for (int i = 0 ; i < 4 ; ++i) {
  if((oldValue = aMap.put(myKey, value++)) != null) {
    System.out.println("Uh-oh, we bounced an object: " + oldValue);
  }
}
```

Of course, you could throw your own exception here instead of displaying a message on the command line. The second parameter to the `put()` method for the `aMap` object is of type `Integer`, so the compiler supplies an autoboxing conversion for the `int` value that is passed as the argument.

If you execute this fragment, it generates the following output:

```
Uh-oh, we bounced an object: 12345
Uh-oh, we bounced an object: 12346
Uh-oh, we bounced an object: 12347
```

When the first value is stored, there's nothing stored in the map for the key, so there's no message. For all subsequent attempts to store objects, the previous object is replaced, and a reference to it is returned.

Note that the `get()` operation returns a reference to the object associated with the key, but it does not remove it from the table. To retrieve an object and delete the entry containing it from the table, you must use the `remove()` method. This removes the object corresponding to the key and returns a reference to it:

```
int objectValue = aMap.remove(myKey);
```

As noted previously, if there's no stored object corresponding to `myKey`, or `null` was stored as the object, `null` is returned. If you were to append this statement to the previous fragment, a reference to an `Integer` object encapsulating the value 12348 would be returned. Because you store it in a variable of type `int`, the compiler inserts an unboxing conversion for the return value.

Processing All the Elements in a Map

The `Map<>` interface provides three ways of obtaining a collection view of the contents of a map. You can obtain all the keys from a `Map<K,V>` object as an object of type `Set<K>`.You can also get a `Collection<V>` object that contains all the values in the map. Key/object pairs are stored in a map as objects of a type that implements the `Map.Entry<K,V>` interface. This is a generic interface type that is defined within the `Map<K,V>` interface. You can get all the key/object pairs from the map as an object of type `Set<Map.Entry<K,V>>`.

Note that the `Set<>` or `Collection<>` object that you get is essentially a view of the contents of a map, so changes to a `HashMap<>` object are reflected in the associated `Set<>` or `Collection<>`, and *vice versa*. The three methods involved are the following:

- `Set<K> keySet()`: Returns an object containing all the keys from the map.
- `Set<Map.Entry<K,V>> entrySet()`: Returns an object containing the key/object pairs—each pair being an object of type `Map.Entry<K,V>`.
- `Collection<V> values()`: Returns an object containing all the values stored in the map.

Let's first see how you can use a set of keys. The `keySet()` method for a `HashMap<K,V>` object returns a `Set<K>` object containing the set of keys that you can either use directly to access the keys or use indirectly to get at the objects stored in the map. For a `HashMap<String, Integer>` object `aMap`, you could get the set of all the keys in the map with the following statement:

```
Set<String> keys = aMap.keySet();
```

Now you can get an iterator for this set of keys with this statement:

```
Iterator<String> keyIter = keys.iterator();
```

You can use the `iterator()` method for the object `keys` to iterate over all the keys in the map. Of course, you can combine these two operations to get the iterator directly. For example:

```
Iterator<String> keyIter = aMap.keySet().iterator();  // Get the iterator

while(keyIter.hasNext()) {                            // Iterate over the keys
  System.out.println(keyIter.next());
}
```

This iterates over all the keys and outputs them.

The `Set<>` interface has `Iterable<>` as a superinterface, so you can use the collection-based `for` loop directly with the object that the `keySet()` method returns:

```
for(String key : aMap.keySet()) {
  System.out.println(key);
}
```

That's much neater than messing about with an iterator, isn't it? In general, the collection-based `for` loop provides you with code that is easier to understand than an iterator.

Of course, you could use the keys to extract the values but the `Collection<>` object that is returned by the `values()` method provides you with a more direct way of doing this. Here's how you could list the values stored in `aMap`, assuming it is of type `HashMap<String,Integer>`:

```
Collection<Integer> collection = aMap.values();
for(Integer i : collection) {
  System.out.println(i);
}
```

This uses a collection-based `for` loop to iterate over the elements in the collection of values that the `values()` method returns.

The `entrySet()` method returns a `Set<Map.Entry<K,V>>` object containing the key/object pairs. In a similar way to that used for the set of keys, you use a `for` loop to access the `Map.Entry<>` objects. Each `Map.Entry<K,V>` object contains the following methods:

- `K getKey()`: Returns the key for the `Map.Entry<K,V>` object.
- `V getValue()`: Returns the value for the `Map.Entry<K,V>` object.
- `V setValue(V new)`: Sets the value for this `Map.Entry<K,V>` object to the argument and returns the original value. Remember that this alters the original map. This method throws:
  - `UnsupportedOperationException` if `put()` is not supported by the underlying map.
  - `ClassCastException` if the argument cannot be stored because of its type.
  - `IllegalArgumentException` if the argument is otherwise invalid.
  - `NullPointerException` if the map does not allow `null` objects to be stored and the argument is `null`. This last exception does not apply to `HashMap<>`.

A `Map.Entry<>` object also needs an `equals()` method for comparisons with another `Map.Entry<>` object passed as an argument and a `hashCode()` method to compute a hashcode for the `Map.Entry` object. With a set of `Map.Entry<>` objects you can obviously access the keys and the corresponding values using a collection-based `for` loop, and you can modify the value part of each key/value pair if you need to.

You have waded through a lot of the theory for `HashMap<>` objects; let's put together an example that applies it.

You can create a very simple phone book that uses a map. We won't worry too much about error recovery so as not to bulk up the code. You'll use a variation of the last version of the `Person` class that you saw earlier in this chapter in the example where you were sorting objects in a vector. Copy the source file to a new directory called `TryPhoneBook` or something similar. Besides the `Person` class, you need to create a `PhoneNumber` class and a `BookEntry` class that represents an entry in your phone book combining a name and a number. You could add other stuff such as the address, but this is not necessary to show the principles. You'll also define a `PhoneBook` class to represent the phone book.

TRY IT OUT Using a HashMap Map

You need to improve your old `Person` class to make `Person` objects usable as keys in the map that you use—to store the phone book entries. You must add an `equals()` method to do this, and you override the default `hashCode()` method just to show how this can work. The extended version of the class is as follows:

Available for download on Wrox.com

```
import java.io.*;

public class Person implements Comparable<Person>, Serializable {
  @Override
  public boolean equals(Object person) {
    return compareTo((Person)person) == 0;
  }

  @Override
  public int hashCode() {
    return 7*firstName.hashCode()+13*surname.hashCode();
  }

  // The rest of the class as before...
  private static final long serialVersionUID = 1001L;
}
```

Directory "TryPhoneBook 1"

You've added to the previous version of the class two methods that override the `equals()` and `hashCode()` methods inherited from `Object` and defined `serialVersionUID` for the class. Because the `String` class defines a good `hashCode()` method, you can easily produce a hash code for a `Person` object from the data members. To implement the `equals()` method you just call the `compareTo()` method that you implemented for the `Comparable<>` interface. You have also made the class serializable just in case it comes in useful at some point.

There's another thing you can do that is definitely useful. You can add a `static` method to the `Person` class that reads data for a `Person` object from the keyboard:

Available for download on Wrox.com

```
import java.io.*;

public class Person implements Comparable<Person>, Serializable {
  // Read a person from the keyboard
  public static Person readPerson() {
    String firstName = null;
    String surname = null;
    try {
      System.out.print("Enter first name: ");
      firstName = keyboard.readLine().trim();
      System.out.print("Enter surname: ");
      surname = keyboard.readLine().trim();
    } catch(IOException e) {
      System.err.println("Error reading a name.");
      e.printStackTrace();
      System.exit(1);
    }
    return new Person(firstName,surname);
  }

  // Rest of the class as before...
  private static BufferedReader keyboard = new BufferedReader(
                                       new InputStreamReader(System.in));
}
```

Directory "TryPhoneBook 1"

You should have no trouble seeing how this works because it's almost identical to the `readPerson()` method that you used previously in this chapter.

You can make the `PhoneNumber` class very simple:

Available for download on Wrox.com

```
import java.io.*;

class PhoneNumber implements Serializable {
  public PhoneNumber(String areacode, String number) {
    this.areacode = areacode;
    this.number = number;
  }

  @Override
  public String toString() {
    return areacode + " " + number;
  }

  private String areacode;
  private String number;
  private static final long serialVersionUID = 1001L;
}
```

Directory "TryPhoneBook 1"

You could do a whole lot of validity checking of the number here, but it's not important for the example.

However, you could use a `static` method to read a number from the keyboard, so let's add that, too:

Available for download on Wrox.com

```
import java.io.*;

class PhoneNumber implements Serializable {
  // Read a phone number from the keyboard
  public static PhoneNumber readNumber() {
    String area = null;                              // Stores the area code
    String localcode = null;                         // Stores the local code
    try {
      System.out.print("Enter area code: ");
      area = keyboard.readLine().trim();
      System.out.print("Enter local code: ");
      localcode = keyboard.readLine().trim();
      System.out.print("Enter the number: ");
      localcode += " " + keyboard.readLine().trim();
    } catch(IOException e) {
      System.err.println("Error reading a phone number.");
      e.printStackTrace();
      System.exit(1);
    }
    return new PhoneNumber(area,localcode);
  }

  // Rest of the class as before...
  private static BufferedReader keyboard = new BufferedReader(
                                       new InputStreamReader(System.in));
}
```

Directory "TryPhoneBook 1"

This is again very similar to the `readPerson()` method. You don't need a separate variable to store the number that is entered. You just append the string that is read to `localcode`, with a space character inserted to make the output look nice. In practice, you certainly want to verify that the input is valid, but you don't need this to show how a hash map works.

An entry in the phone book combines the name and the number and probably includes other things such as the address. You can get by with the basics:

Available for download on Wrox.com

```
import java.io.Serializable;

class BookEntry implements Serializable {
  public BookEntry(Person person, PhoneNumber number) {
   this.person = person;
   this.number = number;
  }

  public Person getPerson() {
    return person;
  }

  public PhoneNumber getNumber() {
    return number;
  }

  @Override
  public String toString() {
    return person.toString() + '\n' + number.toString();
  }

  // Read an entry from the keyboard
  public static BookEntry readEntry() {
    return new BookEntry(Person.readPerson(), PhoneNumber.readNumber());
  }

  private Person person;
  private PhoneNumber number;
  private static final long serialVersionUID = 1001L;
}
```

Directory "TryPhoneBook 1"

This is all pretty standard stuff. In the `static readEntry()` method, you just make use of the methods that create `Person` and `PhoneNumber` objects using input from the keyboard, so this becomes very simple.

The class that implements the phone book is next—called the `PhoneBook` class, of course:

Available for download on Wrox.com

```
import java.io.Serializable;
import java.util.HashMap;

class PhoneBook implements Serializable {
  public void addEntry(BookEntry entry) {
    phonebook.put(entry.getPerson(), entry);
  }

  public BookEntry getEntry(Person key) {
    return phonebook.get(key);
  }

  public PhoneNumber getNumber(Person key) {
    BookEntry entry = getEntry(key);
    if(entry != null) {
    return entry.getNumber();
```

```
      } else {
        return null;
      }
    }

    private HashMap<Person,BookEntry> phonebook = new HashMap<>();
    private static final long serialVersionUID = 1001L;
  }
```

Directory "TryPhoneBook 1"

To store `BookEntry` objects you use a `HashMap<Person,BookEntry>` member, `phonebook`. You use the `Person` object corresponding to an entry as the key, so the `addEntry()` method has to retrieve only the `Person` object from the `BookEntry` object that is passed to it and use that as the first argument to the `put()` method for `phonebook`.

All you need now is a class containing `main()` to test these classes:

Available for download on Wrox.com

```
public class TryPhoneBook {
  public static void main(String[] args) {
    PhoneBook book = new PhoneBook();                  // The phone book
    FormattedInput in = new FormattedInput();          // Keyboard input
    Person someone;
    while(true) {
      System.out.println("Enter 1 to enter a new phone book entry\n" +
                         "Enter 2 to find the number for a name\n" +
                         "Enter 9 to quit.");
      int what = 0;                                   // Stores input selection
      try {
        what = in.readInt();

      } catch(InvalidUserInputException e) {
        System.out.println(e.getMessage() + "\nTry again.");
        continue;
      }

      switch(what) {
        case 1:
          book.addEntry(BookEntry.readEntry());
          break;
        case 2:
          someone = Person.readPerson();
          BookEntry entry = book.getEntry(someone);
          if(entry == null) {
            System.out.println(
                            "The number for " + someone + " was not found.");
          } else {
            System.out.println(
                     "The number for " + someone + " is " + entry.getNumber());
          }
          break;
        case 9:
          System.out.println("Ending program.");
          return;
        default:
          System.out.println("Invalid selection, try again.");
          break;
      }
    }
  }
}
```

Directory "TryPhoneBook 1"

You're using the `FormattedInput` class that you developed in Chapter 8 to read the input values, so copy the source file for this class along with the source file for the `InvalidUserInputException` class, which is also from Chapter 8, to the directory for this example.

This is what the example produces with my input:

```
Enter 1 to enter a new phone book entry
Enter 2 to find the number for a name
Enter 9 to quit.
1
Enter first name: Algernon
Enter surname: Lickspittle
Enter area code: 914
Enter local code: 321
Enter the number: 3333
Enter 1 to enter a new phone book entry
Enter 2 to find the number for a name
Enter 9 to quit.
2
Enter first name: Algernon
Enter surname: Lickspittle
The number for Algernon Lickspittle is 914 321 3333
Enter 1 to enter a new phone book entry
Enter 2 to find the number for a name
Enter 9 to quit.
9
Ending program.
```

Of course, you can try it with several entries if you have the stamina.

How It Works

The values in the map that represents a phone book are `BookEntry` objects to allow for more information to be stored about a person. If you wanted to keep it really simple, you could use `PhoneNumber` objects as values in the map.

The `main()` method runs an ongoing loop that continues until a `9` is entered. When a `1` is entered, the `addEntry()` method for the `PhoneBook` object is called with the expression `BookEntry.readEntry()` as the argument. The `static` method `readEntry()` calls the `static` methods in the `Person` class and the `PhoneNumber` class to read from the keyboard and create objects of these classes. The `readEntry()` method then passes these objects to the constructor for the `BookEntry` class, and the object that is created is returned. This object is added to the `HashMap` member of the `PhoneBook` object.

If a `2` is entered, the `getEntry()` method is called. The `readPerson()` member of the `Person` class is called to obtain the `Person` object corresponding to the name entered from the keyboard. This object is then used to retrieve an entry from the map in the `PhoneBook` object. Of course, if there is no such entry `null` is returned, so you have to check for it and display an appropriate message.

`String` comparisons are case sensitive so searching for "algernon lickspittle" will not find an entry. You could convert both strings to the same case if you wanted to have comparisons that were not case sensitive.

TRY IT OUT Storing a Map in a File

This phone book is not particularly useful. The process of echoing what you just keyed in doesn't hold one's interest for long. What you need is a phone book that is held in a file. That's not difficult. You just need to add a constructor and another method to the `PhoneBook` class:

Available for download on Wrox.com

```
import java.nio.file.*;
import java.io.*;
import java.util.*;

class PhoneBook implements Serializable {
  public PhoneBook() {
    if(Files.exists(file)) {                  // If there's a phone book in a file...
      try (ObjectInputStream in = new ObjectInputStream(
                        new BufferedInputStream(Files.newInputStream(file)))){
        phonebook = (HashMap<Person, BookEntry>)in.readObject();   //...read it in.

      } catch(ClassNotFoundException| IOException e) {
        e.printStackTrace();
        System.exit(1);
      }
    }
  }

  public void save() {
    try {
      Files.createDirectories(file.getParent()); // Make sure we have the directory
    } catch (IOException e) {
      System.err.println("I/O error creating directory. " + e.getMessage());
      e.printStackTrace();
      System.exit(1);
    }
    try (ObjectOutputStream out = new ObjectOutputStream(
                            new BufferedOutputStream(Files.newOutputStream(file)))){
      System.out.println("Saving phone book");
      out.writeObject(phonebook);
      System.out.println("Done");
    } catch(IOException e) {
      System.err.println("I/O error saving phone book. " + e.getMessage());
      e.printStackTrace();
      System.exit(1);
    }
  }
  // Other members of the class as before...

  private Path file = Paths.get(System.getProperty("user.home")).
                        resolve("Beginning Java Stuff").resolve("Phonebook.bin");}
```

Directory "TryPhoneBook 2"

The new private data member `file` defines the path for the file where the map holding the phone book entries is to be stored. The `file` object is used in the constructor that now reads the `HashMap<>` object from the file if it exists. If it doesn't exist, the constructor does nothing, and the `PhoneBook` object uses the default empty `HashMap` object. The cast of the reference returned by the `readObject()` method to type `HashMap<Person, BookEntry>` causes the compiler to issue a warning message to the effect that you have an unchecked cast. There is no way around this because the compiler cannot know what the type of the object that is read from the file is. Everything is fine as long as you know what you are doing!

The `save()` method provides for storing the map away, so you need to call this method before ending the program.

To make the program a little more interesting you could add a method to the `PhoneBook` class that lists all the entries in a phone book. Ideally, the entries should be displayed in alphabetical order by name. One way to do this would be to create a linked list containing the entries and use the static `sort()` method that the `Collections` class defines to sort them. The `sort()` method expects an argument that is of type `List<>`, where the type of elements in the list implements the `Comparable<>` interface. Thus, to be able to sort the entries in the phone book, the `BookEntry` class must implement the `Comparable<>` interface. This is quite easy to arrange:

Available for download on Wrox.com

```
import java.io.Serializable;

class BookEntry implements Comparable<BookEntry>, Serializable {
  public int compareTo(BookEntry entry) {
    return person.compareTo(entry.getPerson());
    }
    // Rest of the class as before...
}
```

Directory "TryPhoneBook 2"

When sorting the entries, you want the sort order of the `Person` objects to determine the sort order of the `BookEntry` objects. Because the `Person` class already implements the `Comparable<>` interface, you can implement the `compareTo()` method in the `BookEntry` class by calling the method for the `Person` object in the entry.

Now you can implement the `listEntries()` method in the `PhoneBook` class to list the entries in alphabetical order:

Available for download on Wrox.com

```
import java.nio.file.*;
import java.io.*;
import java.util.*;
class PhoneBook implements Serializable {
  // List all entries in the book
  public void listEntries() {
    // Get the entries as a linked list
    LinkedList<BookEntry> entries = new LinkedList<>(phonebook.values());
    Collections.sort(entries);                         // Sort the entries

    for(BookEntry entry : entries) {
      System.out.println(entry);
    }
  }
  // Other members as before...
}
```

Directory "TryPhoneBook 2"

Listing the entries in name sequence is relatively simple. Calling the `values()` method for the phonebook object returns the objects in the map, which are `BookEntry` objects, as a `Collection<>`. You pass this to the constructor for the `LinkedList<BookEntry>` class to obtain a list of entries. The `LinkedList<>` class implements the `List<>` interface, so you can pass the `entries` object to the `sort()` method to sort the values. It's then a simple matter of using the collection-based `for` loop to iterate through the sorted values to output them.

You can update `main()` to take advantage of the new features of the `PhoneBook` class:

Available for download on Wrox.com

```
public class TryPhoneBook2 {
  public static void main(String[] args) {
    PhoneBook book = new PhoneBook();                 // The phone book
    FormattedInput in = new FormattedInput();         // Keyboard input
    Person someone;

    while(true) {
      System.out.println("Enter 1 to enter a new phone book entry\n"+
                         "Enter 2 to find the number for a name\n"+
                           "Enter 3 to list all the entries\n" +
                         "Enter 9 to quit.");
      int what = 0;
      try {
        what = in.readInt();

      } catch(InvalidUserInputException e) {
        System.out.println(e.getMessage() + "\nTry again.");
        continue;
      }

      switch(what) {
        case 1:
            book.addEntry(BookEntry.readEntry());
            break;
          case 2:
            someone = Person.readPerson();
            BookEntry entry = book.getEntry(someone);
            if(entry == null) {
              System.out.println(
                               "The number for " + someone + " was not found.");
            } else {
              System.out.println(
                     "The number for " + someone + " is " + entry.getNumber());
            }
            break;
          case 3:
            book.listEntries();
            break;
          case 9:
            book.save();
            System.out.println("Ending program.");
            return;
          default:
            System.out.println("Invalid selection, try again.");
            break;
        }
      }
    }
  }
```

Directory "TryPhoneBook 2"

How It Works

The first changes here are an updated prompt for input and a new case in the `switch` to list the entries in the phone book. The other change is to call the `save()` method to write the map that stores the phone book to a file before ending the program.

WARNING *Beware of the default* `hashCode()` *method in the* `Object` *class when storing maps in a file. As you know, the hashcodes are generated from the address of the object, so the hashcode for a key is entirely dependent on where it is stored in memory. Getting a key back from a file in exactly the same place in memory where it was originally stored is about as likely as finding hairs on a frog. The result is that when you read a map back from a file, the hashcode generated from a key you now use to access the map is different from what was originally produced when you stored the object with the key, so you will never find the entry in the map to which it corresponds.*

There is a solution to the problem. You must override the default `hashCode()` *method so that the hashcodes for keys are produced from the data members of the key objects. This ensures that the hashcode for a given key is always the same. The* `Person` *class does exactly this by overriding the* `hashCode()` *method.*

The first time you run `TryPhoneBook2` it creates a new file and stores the entire phone book in it. On subsequent occasions the `PhoneBook` constructor reads from the file, so all the previous entries are available.

In the next chapter you move on to look at some of the other components from the `java.util` package.

SUMMARY

All of the classes in this chapter will be useful sooner or later when you're writing your own Java programs. You apply many of them in examples throughout the remainder of the book. Collection classes have been implemented in other languages such as C++, so the concepts you have learned here about ways of storing and organizing objects applicability are applicable in other situations.

EXERCISES

You can download the source code for the examples in the book and the solutions to the following exercises from `www.wrox.com`.

1. Implement a version of the program to calculate prime numbers that you saw in Chapter 4 to use a `Vector<>` object instead of an array to store the primes. (Hint: Remember the `Integer` class.)
2. Write a program to store a deck of 52 cards in a linked list in random sequence using a `Random` class object. You can represent a card as a two-character string — `"1C"` for the ace of clubs, `"JD"` for the jack of diamonds, and so on. Output the cards from the linked list as four hands of 13 cards.
3. Extend the program from this chapter that used a map to store names and telephone numbers such that you can enter a number to retrieve the name.
4. Implement a phone book so that just a surname can be used to search and have all the entries corresponding to the name display.

▶ WHAT YOU LEARNED IN THIS CHAPTER

TOPIC	CONCEPT
Collections Framework	The Java collections framework provides you with a range of collection classes implemented as generic types. These enable you to organize your data in various ways.
`EnumSet<E>`	The `EnumSet<>` collection class enables you to aggregate one or more constants from a given enumeration type into a set so they can be passed to a method as a single argument.
`Vector<T>`	You can use a `Vector<>` object as a kind of flexible array that expands automatically to accommodate any number of objects stored.
`ArrayList<T>`	An `ArrayList<>` is very similar to a vector. The primary difference between them is that a `Vector<>` is thread-safe, whereas an `ArrayList<>` is not.
`Stack<T>`	The `Stack<>` class is derived from the `Vector` class and implements a pushdown stack.
`HashMap<K, V>`	The `HashMap<>` class defines a hash map in which objects are stored based on an associated key.
Iterators	An `Iterator<>` is an interface for retrieving objects from a collection sequentially. An `Iterator<>` object enables you to access all the objects it contains serially—but only once. There's no way to go back to the beginning.
List Iterators	The `ListIterator<>` interface provides methods for traversing the objects in a collection backward and forward.
Using Iterators	Objects stored in any type of collection can be accessed using `Iterator<>` objects.
Using List Iterators	Objects stored in a `Vector<>`, a `Stack<>`, or a `LinkedList<>` can be accessed using `ListIterator<>` objects.